CS330: Collected Lecture Notes

Steven J Zeil

April 18, 2013

Contents

Part I

Course Overview

Chapter 1

Course Structure and Policies

Getting Started

Website: `http://www.cs.odu.edu/~cs330.html`

Syllabus: All students are responsible for reading the course syllabus and abiding by the policies described there.

Details related to the use of the course website and to requirements for assignments and projects can be found on the Policies page. All students are expected to read these before the first assignment is issued.

. . . .

1.1 Course Structure

1.1.1 Readings

Readings

- Chapters from text

- Horstmann, Object-Oriented Design and Patterns, 2nd ed. (required)
- Fowler, UML Distilled, 3rd Ed. (optional)

- Online Lecture Notes
- Other Online Materials
  - Oracle tutorials
  - CS 382

. . . .

1.1.2 Assignments

Assignments

- Programming assignments
  - Individual work
  - Focused on current lesson
  - Usually graded automatically - results available in 30 min or less
- Analysis assignments
  - Recommended to work in teams
  - Explore the process of moving from vague understanding of a problem toward design
- Roughly speaking, programming and analysis assignments will alternate
  - dates may overlap

. . . .

1.1.3 Exams

Exams

- Midterm & Final
  - Administered on-line
  - Dates & times TBA
- Final exam is cumulative.

. . . .

1.2 Computer Access

Computer Access

- *Top priority* – make sure that you have a valid CS account - do it this week!
- Available machines:
  - CS Dept labs in Dragas and E&CS
  - Virtual PC lab – see CS Dept home page
  - Unix network (Linux machines - `linux.cs.odu.edu`)
    - some assignments *will* require use of this

. . . .

1.3 Related Courses

Related Courses

- CS 250 or CS 333: Programming in C++

  Prerequisite for this course.

  - If you need review, the CS 333 site is a good choice
  - If you are not used to working with our PC network, pay special attention to the Labs
- CS 252: Introduction to Unix for Programmers

  Prerequisite for this course
- CS361: Advanced Data Structures & Algorithms
- CS 382: Introduction to Java
  - Relatively new 1-credit course
  - Most of the reading material & labs from that course will be required in this one (in lieu of a Java textbook)
- CS 350: Software Engineering
  - Changes in 350 include moving some material from 330 into 350

. . . .

1.4 Important Policies

Late Submissions

. . . are not normally accepted. Exceptions may be made in cases of

- documented emergencies

- arranged prior to the due date when possible.

Extensions to due dates will *not* be granted due to

- difficulties "porting" from one system to another
- transient system crashes
- system overloaded

. . . .

Academic Honesty

ODU is governed by a student honor code.

- Everything you turn in for grading must be your own work.
- Detailed policy statement is in the syllabus.

. . . .

Academic Honesty (cont.)

- Aiding a fellow student to copy someone else's work (including your own) places you equally in violation.
  - Includes leaving work world-readable on the computer system!
- Failure to report observed violations of the honor code is also a violation.

. . . .

Grading

Assignments:	40%
Midterm Exam:	25%
Final Exam:	35%

- Expect a short assignment roughly every 2 weeks. (6-7 total)
  - Most of these will be programming assignments;
  - Two will be analysis/design assignments.
- Programming assignments are graded automatically when possible

. . . .

Chapter 2

Main Course Themes

This course is concerned with object-oriented programming (OOP), design (OOD), and analysis (OOA). Clearly, with so much of this OO stuff flying around, it would be nice if we had an idea of what an object is, and just what is meant by "object-oriented".

2.1 Leading Up to OO

2.1.1 Observations on Pre-OO Programming Languages

The Roots of OOP

- OO programming and design have their roots in simulation.
- Simula (1967) programming language introduced objects & classes to describe the physical world being simulated.

. . . .

OO programming and design have their roots in simulation. In the late 1960's, the population of programmers was dominated by people who had switched fields, typically from the mathematics or physical sciences, rather than people who had originally trained in computer science. Indeed, academic programs in computer science were far less common than they are today, and tended to treat CS as more of a tool to be applied to application areas rather than as a field of study in its own right. Commonly, a CS degree program would offer two tracks - business programming and scientific programming.

One of the more active and exciting application areas at the time was simulation. Programmers in this area began to recognize common patterns arising over and over in their code. A number of code libraries were designed and distributed that attempted to capture and simplify these common simulation tasks. Eventually, special-purpose simulation programming languages were developed.[1]

Simula (1967) is now the best known of those special purpose simulation languages. In what seemed "natural" to someone writign simulations, Simula allowed a programmer to organize a simulation program in terms of a world of objects that interacted with one another via programmed behaviors. What could seem more natural to someone writing a simulation of some part of the real world?

Tension in Pre-OO Programming Languages

The history of programming languages and of design can be viewed as a continual contest against increasing complexity of software systems:

Problem: too many…	**Response**	
… statements	nesting ({…})	OK
	gather statements into functions	
… functions	nesting	(inadequate)
	gather functions into subsystems / "modules"	Too loosely defined
	gather functions & data into encapsulated ADTs	(Pre-dates OOPs by more than a decade)
… ADTs	Gather into namespaces/packages	Not much help
	Organize loosely into inheritance hierarchies	The OO approach

Meanwhile, back in the "main stream" of programming language developement, one of the main historic trends has always been trying to keep programs well organized even as programs get larger and larger. The earliest and simplest way to cope with size was to allow individual statements to be grouped together into functions/procedures/subroutines (all effectively meaning the same thing). In fact this idea pre-dates high-level programming languages, being implicit in the machine code instruction

[1] There was ample precedence for this kind of specialization. The dominant languages of the time were generally perceived as tailored to specific applications: FORTRAN and APL for science applications, COBOL for business, LISP for A.I., SNOBOL for text/string processing, etc. Any of these languages *could* have been used for other kinds of applications, but it was certainly easier to work with a language that matched well with the application area.

sets of almost all CPUs ever built. It may even pre-date any actual programming. Some attribute this idea to Ada Lovelace's programs for Babbage's never-constructed analytical engine.

Functions allow us to cope when the number of individual statements in a program grows so large as to be unwieldy. So what do we do when the number of functions grows too large to be manageable? Programmers and designers tried to cope by grouping functions into "modules" or "subsystems". These groupings were little more than a matter of documentation and physical arrangement of the functions within the source code files. On a team project, a single person who did not want to "play nice" could easily wreak havoc.

A more solid organizing principle arose in the notion of an Abstract Data Type (ADT). We'll review this concept in more detail in a later lesson, but an ADT groups together a data type name and a collection of functions for manipulating data of that type.

ADTs were something of a revolution in programming, as they often led to remarkably clean, reusable modules. Of course, like all modules, the ADT originally was implemented simply by documenting it as such: "the functions in this file provide the ADT Stack..." An uncooperative team member could often compromise this organization (usually giving plausible but short-sighted excuses such as "I could get the program done faster if I bypassed that" or "It runs a millisecond faster if I access that bit of data directly instead of using those functions").

The value of ADTs became widely accepted, and programming language designers responded by adding support for them into new programming languages. Languages such as Modula 2 (1970) and Ada (1983) added language mechanisms (called a "module" and a "package", respectively) for grouping types and functions together. Both enforced "information hiding" by encapsulating portions of an ADT implementation that other programmers would then be unable to access without incurring compilation errors.

Many writers incorrectly regard ADTs, encapsulation, and information hiding as innovations of OOP, but in fact these ideas had been around for some time earlier, and were in widespread use before object-oriented programming languages (OOPL) made their entrance.

Indeed, we will eventually see that the vast majority of code written by even the very best programmers working in an OOPL is not particularly OO, but *will* exploit encapsulated ADTs very heavily.

2.1.2 Pre-OO Design Techniques

Observations on Pre-OO Design Techniques

- Google for ADTs.
    - Most of the examples you will get are general-purpose data structures (stacks, vectors, lists, etc)

  - It's easy to get the impression that an ADT is something that someone else provides for you as a convenient little package of code.
  - Misses the big point

- ADTs are an *organizing principle* for your own programs.
- But how do you design a program in a way that winds up with good ADTs?

. . . .

OK, let's agree that a good ADT is a wonderful thing. Just where do good ADTs come from? Or, to put it another way, when you are designing a program, how do you figure out which ADTs you should use?

If you search the web for the term "abstract data type", you will find lots of examples of things you might pull out of a library of data structures: stacks, queues, lists, etc. It's easy to get the impression that an ADT is something that someone else provides for you as a convenient little package of code.

But that misses the point that "abstract data type" is an *organizing principle* for your own programs. ADTs for general purpose data structures are all very nice, but how do you come up with the special-purpose ADTs that organize your program in a way that is specific to the problem it solves?

Pre-OO design techniques were oriented towards functions performed by the program. For example, if you were working on automating the management of book handling in a typical library, you might start by dividing the system into major subsystems such as *RecordStorage*, *CheckInOut*, *AccountMgmt*, and *InventoryMgmt*. Under each subsystem, you would group functions to be performed, such as *CheckOutBook*, *CheckInBook*, and *RenewBook* under the *CheckInOut* subsystem, or *addBook*, *removeBook*, and *transferToBranch* under *InventoryMgmt*.

Stepwise Refinement & Top-Down Design

- The most fundamental design technique

. . . .

Stepwise Refinement Example

Automating book handling in a library:

- Might start by dividing the system into major subsystems such as `RecordStorage`, `CheckInOut`, `AccountMgmt`, and `Inventory`
  Under each subsystem, you would group functions to be performed, such as `CheckOutBook`, `CheckInBook`, and `RenewBook` under the `CheckInOut` subsystem, or `addBook`, `removeBook`, and `transferToBranch` under `InventoryMgmt`.

. . . .

If pre-OO designers were asked to design one of these functions, say, *addBook*, they would probably do so by "stepwise refinement", a.k.a. "top-down design".

Stepwise Refinement Example II

They would reason that this function must both update the electronic card catalog so that the book can be found, and must also record that the book is present in the inventory of a particular branch:

```
function addBook (book, branchName)
{
  bookInfo = getBookInfo(book);
  record branchName as location in bookInfo;
  addToCardCatalog (bookInfo);
  addToInventory (bookInfo, branchName);
}
```

. . . .

Stepwise Refinement Example III

Next the designers would pick one of those rather vaguely understood lines of code in that function body and expand it, either in place or as a separate function. For example:

```
function addBook (book, branchName)
{
  bookInfo = getBookInfo(book);
  record branchName as location in bookInfo;
  // addToCardCatalog (bookInfo)
  add to AuthorIndex (bookInfo);
  add to SubjectIndex (bookInfo);
  add to TitleIndex (bookInfo);
  cardCatalog.addToAuthorIndex
  addToInventory (bookInfo, branchName);
}
```

Then, again, we would pick a vaguely understood line and expand it.

Stepwise Refinement Example IV

```
function addBook (book, branchName)
{
  bookInfo = getBookInfo(book);
  record branchName as location in bookInfo;
  // addToCardCatalog (bookInfo)
  //   add to AuthorIndex (bookInfo);
  authorList = bookInfo.getAuthors();
  for each author au in authorList {
    catalog.authorIndex.addByAuthor (au, bookInfo);
  }
  add to SubjectIndex (bookInfo);
  add to TitleIndex (bookInfo);
  cardCatalog.addToAuthorIndex
  addToInventory (bookInfo, branchName);
}
```

- The process of "pick something vague and expand it" continues until the desired level of detail has been reached.

Top-Down Example: Observations

- In this example, we have mentioned nearly a dozen functions. None of them have been associated with an ADT.

  - An experienced designer might spot some ADT candidates: Book, BranchLibrary, CardCatalog, etc.
  - But there's nothing in the *process* that encourages or supports this.

. . . .

Now, stepwise refinement is a valuable technique. It's still probably the technique of choice for low-level design (the design of an algorithm for a single function). But it's not nearly so useful as a technqiue for high-level design (the separation fo a large problem into separate modules). Consider that, at this point in our example, we have mentioned about a dozen different function names. *None* of them have been associated with an ADT. What are the odds that any of them would wind up inside an ADT? Pretty slim, usually.

An experienced designer might take note that the above process gives some hints as to some potential ADTs (*Book*, *branchLibrary*, *CardCatalog*, etc.). But this insight is pretty much separate from the design process itself. A designer has to be somewhat schizoid, actively participating in the design while simultaneously standing aside as a detached observer, watching for hints of possible ADTs.

2.2 The Object-Oriented Philosophy

2.2.1 Program == Simulation

The Object-Oriented Philosophy

- Every program is really a simulation.
- The quality of a program's design is proportional to the faithfulness with which the structures and interactions in the program mirror those in the real world.

. . . .

If OO analysis and design is about any one thing, it's about how to find and recognize good ADTs. The first proponents of OO looked back to simulation, one of the early success stories in programming and design. Many early simulation programmers were able to construct programs that were organized in a way that made them easy to modify and maintain. This, it was thought, was because the focus on modeling real world, in a language that created a module for each kind of real-world object, led to programs that felt "natural" and easily understood by anyone who understood the part of the real world that was being simulated.

The *Object-Oriented philosophy* states:

- *Every program is really a simulation.*
- *The quality of a program's design is proportional to the faithfulness with which the structures and interactions in the program mirror those in the real world.*

According to this philosophy, a program to manage book handling in a library is really a simulation of what the librarians would have done prior to automation. A program to compute a company payroll is a simulation of what the accounting clerks would have done. A program to compute student grades at the end of a semester is a simulation of what a teacher would have done manually. Now, sometimes the worlds being simulated may be artificial or imaginary (e.g., chess and other games, fractal mathematics) but, as long as they are well understood, that really doesn't matter. You may be able tome with exceptions to the "every program is a simulation" rule, but the very fact that you have to think for a while to do so is probably significant.

If you buy the first part of this philosophy, then the second part becomes really interesting. Ever listen to another programmer explain a design and wonder how the heck that was going to solve the problem at hand? Ever notice how programmers love to make up new terms for things in their code, often in defiance of the normal English-language meaning of the word? Or, even worse, ever read code where the variables had names like *info* or *data* or *value* or the functions had names like *processData*? Does anyone think names like those *really* have any informational content?

The OO philosophy suggests that the things manipulated by the program should correspond to things in the real world. They should carry the same names as those things carry in the real world. And they should interact in ways like those objects n the real world.

Design from the World

- Walk into a library and what do you see? Books, Librarians, Patrons (customers), Shelves, ... In an older library you might actually see a card catalogue!

  Of course, on the way in the door you saw the building/Branch Library.

- These things form a set of candidate ADTs. Next we would explore the very properties and relationships that these things exhibit in real life:

In the OO approach to our earlier library design, we might never arrive at subsystems like *RecordStorage*, *CheckInOut*, *AccountMgmt*, and *InventoryMgmt*. Instead, we would draw upon our knowledge of the world of libraries and say that this world is populated by *Book*s, *CardCatalog*s, *Library*s, *LibraryBranche*s, *Patron*s, etc., and these would become our initial set of ADTs, our modules for the system design. If our personal knowledge of the world of libraries is not quite up to snuff, we can ask the *domain experts*, the librarians and staff and patrons of the library.

Real-World Relationships

- Books have titles, authors, ISBN, ...
- Librarians and Patrons have names. Patrons have library cards with unique identifiers. Most librarians are also patrons.
- Patrons return books. Librarians check in the returned books.
- Librarians check books out for patrons. (At some libraries, patrons can also do their own check-out.) Librarians handle the processing of newly acquired books.
- Although books can be returned to any branch, each book belongs to a specific branch and will, if necessary, be delivered there by library staff.

. . . .

Organizing the World

OO designers would start by organizing these things into ADTs, e.g.:

Librarian
name
assignment: BranchLibrary
checkOut(Book, patron)
checkIn(Book)
acquisition(Book)

Book
title: string
authors: seq of Author
isbn: ISBN
homeBranch: BranchLibrary

Branch Library
inventory: set of Book
stacks: seq of Shelf
staff: set of Librarian
addToInventory(Book)
removeFromInventory(Book)

Catalog
author Index
subject Index
title Index
add (Book, BranchLibrary)
search(filedName: String, value: String) : seq of CatalogEntry

Simulate the World's Actions

Only then would OO designers consider the specific steps required to add a book to inventory: (update the electronic card catalog so that the book can be found, and record that the book is present in the inventory of a particular branch:

Simulate the World's Actions

```
function Librarian::acquisition (Book book)
{
  BranchLibrary branch = this.assignment;
  Catalog catalog = mainLibrary.catalog;
  catalog.add (book, branch);
  branch.addToInventiory (book);
}

function Catalog::add (Book book)
{
  authorIndex.add (book);
  titleIndex.add (book);
  subjectIndex.add (book);
}
```

OO Approach - Observations

Technically, does the same thing as the earlier design,but

- Division into ADTs is a natural consequence of the "observational" approach
- More concern with terminology ('acquisition" rather than 'addBook", "patron" rather than "customer", "stacks" rather "shelves"
  - Avoids programmer-isms such as BookInfo
- Separation of concerns is better (e.g., catalog functions are not built into the librarian code)

. . . .

What's the advantage to this?

- *We can talk to the domain experts* (librarians). Because we talk about things in the library world (not in some made-up-by-the-programmer world), their knowledge of how things really work is immediately relevant to our design. Because we use their terminology, we avoid a lot of frustration and wasted time explaining artificial and misleading programmer-ese terms:

  "OK, so the LB database..."

  "The what?"

  "The LB database - that's where we store the information that used to be in the card catalog. OK, the LB database has to store info about more than just books. You have magazines, CDs, DVDs, etc. So the LB database can contain any kind of shelvable item."

  "Shelvable item? You mean any kind of publication, right?"

  "Uh, yeah. We called them shelvable items because they represent anything you can put on a shelf. Anyway, the LB database can..."

- *We can talk to each other* on the development team. Anyone working on a software project eventually has to gain a certain knowledge of the application domain. That knowledge carries with it a whole host of expectations as to what is and is not relevant, how things behave, who does what, etc. If we know that, in the real world, librarians check out books, then when we are hunting for the code responsible for checking out books, we should feel safe in assuming it will be a function of the *Librarian* ADT, and that it will be named something like *checkOutBook*, not some bit of programmer-ese like *changeBookLocationToCustomer*.

Grady Booch suggested the "principle of least surprise" as a guide to designing ADT interfaces. The idea is that, given that we all share a certain common understanding of how things are supposed to work, any surprises we encounter when reading someone's design are invariably unpleasant ones, representing a place where the design deviates from our expectation of how it would be most likely to work. Each such surprise is something that we are going to have to remember later, whenever we try to work with that particular designed component. If the choice of ADTs is "natural" to the application domain, if the names are natural, if the interactions (function calls) between them are natural, we have fewer surprises to cope with and fewer diddly little details likely to go wrong.

- *We can understand* our own work better. Ever go back after a couple of months, read your old code, and wonder "What in the world was I thinking?" That's just another kind of surprise, delayed. Again, if the structure of the program reflects our natural understanding of the world in which the program resides, we should avoid many of these unpleasant surprises.

Hopefully, then it's clear how an OO approach can have a big impact on the design of our programs. Closely related to OOD is OOA. If "design" is all about figuring out *how* to make a program do what we want, "analysis" is about first figuring out *what* we want it to do. But programs don't run in isolation. They interact with objects in the surrounding world. So it should not come as a shock that thinking about the world as a collection of interacting objects helps here too. If real-world librarians check out books for library patrons, then isn't it natural to assume that an automated librarian would check out books for library patrons?

2.2.2 Simulation == Modeling

OO Analysis & Design

- Analysis is the process of determining *what* a system should do
- Design is the process of figuring out how to do that

. . . .

Models

We can rephrase that as

- Analysis is the construction of a model of the world in which the program will run.
  - including how the program will interact with that world
- Design is the construction of a model of a program to work within that world.

. . . .

Note our approach in the earlier library examples: we built a model of the library world, and took for granted that this was the basis for design of library code.

Steps in Modeling

- Classification: discovering appropriate classes of objects
- Refined by documenting
  - interactions between objects
  - relations between classes
- Both steps often driven by scenarios

. . . .

2.3 Putting the "Programming" in OOP

OOP in Programming Language History

Programming Languages		Design
Concepts	**Examples**	
Statements		Stepwise Refinement
Functions	FORTRAN (1957), COBOL, ALGOL, Basic (1963), C (1969), Pascal (1972)	Top-Down Design
Encapsulated modules, classes	Modula, Modula 2 (1970), Ada (1983)	Information hiding, ADTs
Inheritance, subtyping, dynamic binding (OOP)	Smalltalk (1980), C++ (1985), Java (1995)	Object-Oriented A&D

. . . .

So, if the point of OOA and OOD is to come up with good ADTs, why do we need OOP? In fact, many OODs lead to programs that could be implemented very nicely in a pre-OO programming language that had good support for ADTs. But certain patterns of behavior crop up again and again in simulations, patterns that could not be easily supported in those earlier languages. The languages that we call object-oriented evolved to support these behaviors. This does not replace support for ADTs - all OOPLs start with good support for ADTs. But then they add to it the capability for class- or object-variant behaviors.

2.3.1 What is an Object?

Maybe you thought I was never going to get around to this?

Objects

An *object* is characterized by

- identity
- state
- behavior

. . . .

Identity

Identity is the property of an object that distinguishes it from all others. We commonly denote identity by

- names, such as Steve, George,
  - in programming, x, y
- references that distinguish without names: this, that,
  - in programming, *p

. . . .

State

The *state* of an object consists of the properties of the object, and the current values of those properties. For example, given this list of properties:

```
struct Book {
   string title;
   Author author;
   ISBN isbn;
   int edition;
   int year;
   Publisher publisher;
};
```

we might describe the state of a particular book as

```
Book cs330Text = {
   "Object Oriented Design & Patterns", horstmann,
   "0-471-74487-5", 2, 2006,
   wileyBooks};
```

. . . .

Behavior

Behavior is how an object acts and reacts to operations on it, in terms of

- visible state changes and
- operations on other visible objects.

. . . .

OK, identity, state, and behavior are nice ideas. But what's it really add up to? How do you know if you have a *good* object?

Personally, I'm rather fond of the "kick it" test. If you can kick it, it's an object. Returning again to our library example, *Book*s, *CardCatalog*s, *Library*s, *LibraryBranch*es, *Patron*s are all physical tangible objects. If I came up to one, I could give it a kick. (Some of them might kick back!) So I think I can accept all of these as (classes of) objects.

What about *RecordStorage*, *CheckInOut*, *AccountMgmt*, and *InventoryMgmt*. Have you ever walked down a hall and tripped over an InventoryMgmt? Could you kick/touch/hold a CheckInOut? That's a pretty good clue that these are not objects.

Object (and class) names are invariably noun phrases. A good rule of thumb is that tangible objects in the real world make good objects in the simulated world. This includes people. However, not all simulated objects need to be tangible. Events are often objects. Roles played by people in a system are often objects.

Some things that are terrible choices for objects early in the system development become acceptable later on. For example, is *RecordStorage* an object? It could be. It does not require a whole lot of imagination to believe that it would have identity, state, and behavior. The name isn't great – it smacks of programmer-ese. More importantly, there probably isn't one of these in the library right now, except in the form of the card catalog, in which case we are better off with a *cardCatalog* object.

But later on, when we are deep in design, if we make a design decision to unite the card catalog and circulation and inventory records into a database, calling that a *RecordStorage* object might not be quite so awful (though I'd still prefer a more descriptive name). That's because, although this thing does not exist in the old unautomated world, it will exist in the new world that includes our automated system.

2.3.2 Messages & Methods

Messages & Methods

In OOP, behavior is often described in terms of messages and methods.

A *message* to an object is a request for service.

A *method* is the internal means by which the object responds to that request.

. . . .

Differing Behavior

Different objects may respond to the same message via different methods.

Example: suppose I send a message “send flowers to my grandmother” to…

- a florist
- my sister
- a Department secretary
- a tree

. . . .

The florist responds by checking to see if my grandmother’s address is local. if so, he schedules a delivery to her. If not, he consults his directory of affiliated florits to find one in the same city as my grandmother. He contacts that florits, and passes the request to them in exchange for a portion of the fee.

That is the “florist method” for responding to this message.

My sister responds to the message by a very different meother. She googles for a local florist, gets on the phone to them, and passes my request on to them.

It’s a different method of response to the same message, but, either way, Grandma gets her flowers, so we’re all happy.

If I send the same message to a Department secretary, she responds, “Send your own d— flowers. Who do you think I am? Your servant?”. In programming terms, an exception has been thrown or a run-time error has been signaled. Now, that’s not ideal – Grandma didn’t get her flowers – but, still, it’s something I can work with because I can detect the error message and respond to it.

Finally, if I send the same message to a tree, I get no response at all. Trees just don’t accept that message.

In using objects, we must know what messages they will successfully respond to, not necessarily the method of the response. But how can we even know whether a group of objects respond to some common message? All OOPLs provide some way to organize objects according to the messages they accept. Most common is to group them by class.

2.3.3 What is a Class?

What is a Class?

A *class* is a named collection of potential objects.

In the languages we will study,

- all objects are instances of a class, and
- the method employed by an object in response to a message is determined by its class.
- All objects of a given class use the same method in response to similar messages.

. . . .

By now you might start to wonder if objects and classes are really all that new, or if I'm just playing terminology games with you. In fact, all the terms I have introduced in this lesson have direct correspondences to more traditional programming terms:

Are Objects and Classes New?

OOPL	Traditional PL
object	variable, constant
identity	label, name, address
state	value
class	type (almost)
message	function decl
method	function body
passing a message	function call

. . . .

So all the new vocabulary we've been introduced to has existing equivalents in the traditional programmign world.

One advantage to the new vocabulary is that it helps get you in the mindset of thinking of the program as a simulation of actual objects.

Classes versus Types

Although "class" and "type" mean nearly the same thing, they do carry a slightly different idiomatic meaning.

- In conventional PLs, each value (object) is an instance of exactly one type.
- In OOPLs, each object (value) may be an instance of several related classes.

. . . .

Although you can interchange "type" and "class" in many statements, the use of the word "type" carries with it an implication that only one is involved.

How is it possible for one object to be a member of multiple classes? It happens in the real world all the time that we divide things into multiple levels of ever-finer classes: This occurs because classes can be related via “inheritance”.

Instances of Multiple Types

- Classes can be related via “inheritance”.
  - Inheritance captures an "is-a" or "is a specialized form of of"relationship.

. . . .

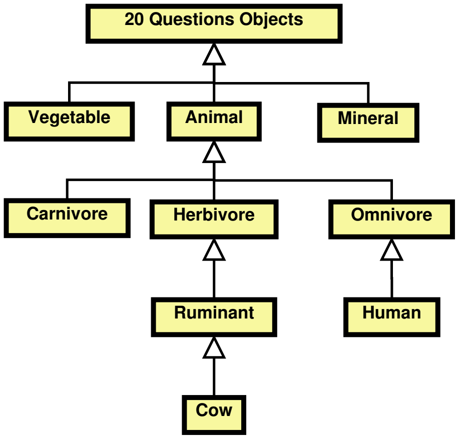
If you have ever played the game “20 Questions”, you may be familiar with the stereotypical opening questions:

“Is it an animal?”

If the answer is no, this is followed with “Is it a vegetable?” (meaning any plant at all). If the answer to that is false, the object is known to be a “mineral” because only tangible objects are considered fair game in most versions of 20 Questions.

If the answer to the animal question were “yes”, however, this is generally followed by questions regarding its diet: “Is it a carnivore?”, “Is it an herbivore?” and so on. An answer of “yes” to the herbivore question might be followed by “Does it eat grass?” (ruminants).

Inheritance Example



. . . .

The diagram here shows that we are progressively restricting ourselves to more specialized classes. Now, clearly Bessie the cow (a specific object) is a member fo the class *Cow* but that is nto all. She is simultaneously a member of the class *Ruminant* and of the classes *Herbivore*, *Animal*, and of the class of valid 20 Questions objects.

This ability to model some classes ans specializations of others is, together with the ability to implement variant behaviors in response to a common message, precisely what traditional PLs were unable to support but that OOPLs were designed to allow.

How do we know if a group of objects will respond to a message? If they are all members, at some level, of a class that supports that message. (Later we will look at an additional mechanism often employed for this purpose, “subtyping”.)

How does an object determine what method to use in response to a message? It uses the method associated with the most specific (specialized) class to which it belongs. We will alter see that this rule is referred to as “dynamic binding”.

What makes a PL an OOPL?

- It must provide support for variant behavior
  - Usually accomplished via *dynamic binding*
- It must provide a way to associate common messages with different classes.

  Usually accomplished via
  - inheritance, and
  - subtyping

. . . .

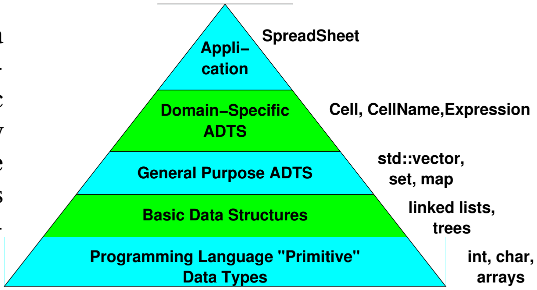
We’ll learn what these more specialized terms mean as the semester progresses.

Part II

Pre-OO: ADTs and C++

Chapter 3

ADTs



If we were to look at a program that is actually large enough to require a full team of programmers to implement it, you would probably not be surprised to find that it would not be organized as a single, large, monolithic unit, but instead as a large number of cooperating functions. You already know how to design and write functions in C++ . What may, however, come as a bit of a surprise if you have not worked much with programs that size is that even these functions will be further organized into higher level structure.

I've tried to illustrate that structure in the diagram that you see here. At the top we have the main application program, for example, a spell checker. The code that occurs at this level is very specific to this application and is the main thing that differentiates a spell checker from, let's say, a spreadsheet. On the other hand, at the very bottom of the hierarchy, we have all the basic primitive data types and operations, such as the int type, the char type, addition, subtraction, and so on, that are provided by our programming language, (C++ in this example). These primitive operations may very well show up in almost any kind of program.

In between those, we have all the things that we can build on top of the language primitives on our way working up towards our application program. Just above the language primitives we have the basic data structures, structures like linked lists or trees. We're going to spend a lot of time the semester looking at these kinds of structures - you may already be familiar with some of them. They are certainly very important. And yet, if we stopped at that level, we would wind up "building to order" for every application. As we move from one application to another we would often find ourselves doing the same kinds of coding, over and over again.

What's wrong with that? Most companies, and therefore most programmers, do not move from one application to a wildly different application on the next project. Programmers who been working on “accounts receivable” are unlikely to start writing compilers the next week, and programmers who have been writing compilers are not going to be writing control software for jet aircraft the week after. Instead, programmers are likely to remain within a general application domain. The people who are currently working on our spell checker may very well be assigned to work on a grammar checker next month, or on some other text processing tool. That means that any special support we can design for dealing with text, words, sentences, or other concepts natural to this application to make may prove valuable in the long run because we can share that work over the course of several projects.

And so, on top of the basic data structures, we expect to find layers of reusable libraries. Just above the basic data structures, the libraries are likely to provide fairly general purpose structures, such as support for look-up tables, user interfaces, and the like. As we move up in the hierarchy, the libraries become more specialized to the application domain in which we're working. Just below the application level, we will find support for concepts that are very close to the spell checker, such as "words", "misspellings", and "corrections".

The libraries that make up all but the topmost layer of this diagram may contain individual functions or groups of functions organized as Abstract Data Types. In this lesson, we'll review the idea of Abstract Data Types and their implementations. Little, if any, of the material in this lesson should be entirely new to you - all of it is covered in CS 250.

3.1 Abstraction

Abstraction

In general,, *abstraction* is a creative process of focusing attention on the main problems by ignoring lower-level details. In programming, we encounter two particular kinds of abstraction:

- *procedural abstraction* and
- *data abstraction*.

. . . .

3.1.1 Procedural Abstraction

Procedural Abstraction

A *procedural abstraction* is a mental model of *what* we want a subprogram to do (but not *how* to do it).

Example: if you wanted to compute the length of the a hypotenuse of a right triangle, you might write something like

```
double hypotenuse = sqrt(side1*side1 + side2*side2);
```

We can write this, understanding that the `sqrt` function is supposed to compute a square root, even if we have no idea how that square root actually gets computed.

- That's because we understand what a square root *is*.

. . . .

When we start actually writing the code, we *implement* a procedural abstraction by

- assigning an appropriately named "function" to represent that procedural abstraction,
- in the "main" code, calling that function, trusting that it will actually do *what* we want but not worrying about *how* it will do it, and
- finally, writing a function body using an appropriate algorithm to do what we want.

In practice, there may be many algorithms to achieve the same abstraction, and we use engineering considerations such as speed, memory requirements, and ease of implementation to choose among the possibilities.

For example, the `sqrt` function is probably implemented using a technique completely unrelated to any technique you may have learned in grade school for computing square roots. On many systems. `sqrt` doesn't compute a square root at all, but computes a polynomial function that was chosen as a good approximation to the actual square root and that can be evaluated much more quickly than an actual square root. It may then refine the accuracy of that approximation by applying Newton's method, a technique you may have learned in Calculus.

Does it *bother* you that `sqrt` does not actually compute via a square root algorithm? Probably not. It shouldn't. As long as we trust the results, the method is something we are happy to ignore.

3.1.2 Data Abstraction

Data Abstraction

Data abstraction works much the same way. A *data abstraction* is a mental model of `what` can be done to a collection of data. It deliberately excludes details of `how` to do it.

. . . .

Example: calendar days

A day (date?) in a calendar denotes a 24-hour period, identified by a specific year, month, and day number.

That's it. That's probably all you need to know for you and I to agree that we are talking about a common idea.

. . . .

Example: cell names

One of the running examples I will use throughout this course is the design and implementation of a spreadsheet. I assume that you are familiar with some sort of spreadsheet program such as Microsoft's Excel or the OpenOffice Calc program. All spreadsheets present a rectangular arrangement of cells, with each cell containing a mathematical expression or formula to be evaluated.

Every cell in a spreadsheet has a unique name. The name has a column part and a row part.

- The row indicators are integer values starting at 1.
- The column indicators are case-insensitive strings of alphabetic characters as follows: `A`, `B`, ..., `Z`, `AA`, `AB`, `AC`, ..., `AZ`, `BA`, `BB`, ... `ZZ`, `AAA`, `AAB`, ... and so on.
- Optional `$` markers may appear in front of each part to "fix" the row or column during copying. For example, suppose we have a cell `B1` containing the formula `2*A1` and that we copy and paste that cell to position C3. When we look in `C3`, we would find the pasted formula was `2*B3` adjusted for the number of rows and columns over which we moved the copied value.

  But if the cell `B1` originally contained the formula `2*A$1`, the copied formula would be `2*B$1`. The `$` indicates that we are fixing the column indicator during copies. Similarly, if the cell `B1` originally contained the formula `2*$A$1`, the copied formula would be `2*$A$1`. (If this isn't clear, fire up a spreadsheet and try it. We can't expect to share mental models (abstractions) if we don't share an experience with the subject domain.)

. . . .

Example: a book

How to describe a book?

- If we are implementing a card catalog and library checkout, it is probably enough to list the metadata
  - (e.g., title, authors, publisher, date).
- If, however, we are going to be working on a project involving the full text of the document (e.g., automatic metadata extraction and indexing), then we might need all the pages and all the text.
- Of course, if we were building bookshelves, we might need more physical attributes such as size and weight!

. . . .

Example: positions within a container

Many of the abstractions that we work with are "containers" of arbitrary numbers of pieces of other data. This is obvious with things like arrays and lists, but is also true of more prosaic items. For example, a book is, in effect, a container of an arbitrary number of authors (and in other variations, an arbitrary number of pages). Any time you have an ordered sequence of data, you can imagine the need to look through it. That then leads to the concept of a *position within that sequence*, with notions like

- finding the first and last position,
- going forward to the next position, etc.

. . . .

3.2 Abstract Data Types

Adding Interfaces

- The *mental model* offered by a data abstraction gives us an informal understanding of how and when to use it.
  - But because it is simply a mental model, it does not tell us enough information to program with it.
- An *abstract data type* (ADT) captures this model in a programming language interface.

. . . .

Definition of an Abstract Data Type

Definition (traditional): An *abstract data type* (ADT) is a type name and a list of operations on that type.
It's convenient, for the purpose of this course, to modify this definition just slightly:
Definition (alternate): An *abstract data type* (ADT) is a type name and a list of members (data or function) on that type.

- an ADT corresponds, more or less, to the public portion of a typical class
- the "list of members" includes
  - names

  - data types
  - expected behavior

- a.k.a. an *ADT specification*

.. This change is not really all that significant. Traditionally, a data member X is modeled as a pair of `getX()` and `putX(x)` functions. But in practice, we will allow ADTs to include data members in their specification. This definition may make it a bit clearer that an ADT corresponds, more or less, to the public portion of a typical class.

In either case, when we talk about listing the members, this includes giving their names and their data types (for functions, their return types and the data types of their parameters).

If you search the web for the phrase “abstract data type”, you’ll find lots of references to stacks, queues, etc. - the “classic” data structures. Certainly, these *are* ADTs. But, just as with the abstractions introduced earlier, each application domain has certain characteristic or natural abstractions that may also need programming interfaces.

ADT Members: attributes and operations

Commonly divided into

- *attributes*: the things that we think of as being data stored inthe ADT
  - Actual interface is often through get*Attr*() and set*Attr*() functions.
  - which, in turn, might or might not actually involve direct access to a "data member"
- operations: the functions or behaviors or the ADT
  - the "type" of a function consists of its return type and an ordered list of of its parameters’ types

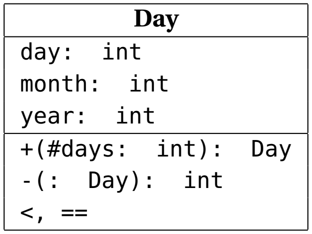
. . . .

3.2.1 Examples

Calendar Days

Nothing in the definition of ADT that says that the interface *has* to be written out in a programming language.

UML diagrams present classes as a 3-part box: name, attributes, & operations



Calendar Days: alternative

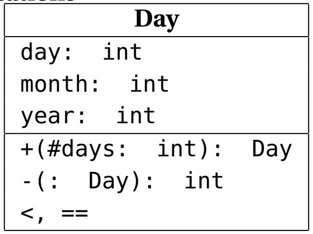
But we can use a more programming-style interface:

```
class Day {
public:
   // Attributes
   int getDay();
   void setDay (int);
   int getMonth();
   void setMonth(int);
   int getYear();
   void setYear(int);

   // Operations
   Day operator+ (int numDays);
   int operator- (Day);
   bool operator< (Day);
   bool operator== (Day);
     ⋮
```

See also the interface developed in sections 3.1 and 3.2 of your text (Horstmann).

- It's essentially the same ADT, but lots of details are different

Notations



```
class Day {
public:
   // Attributes
   int getDay();
   void setDay (int);
  ⋮
```

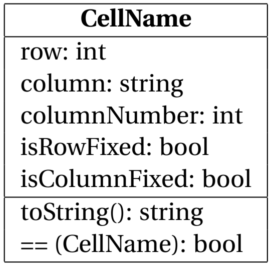
- Disadvantages of moving early to programming-style interfaces:
  - getting lost in language details
  - prematurely committing to those details

. . . .

Cell Names

Here is a possible interface for our cell name abstraction.

```
class CellName
{
public:
  CellName (std::string column, int row,
            bool fixTheColumn = false,
            bool fixTheRow=false);
  //pre: column.size() > 0 && all characters in column are alphabetic
  //     row > 0

  CellName (std::string cellname);
  //pre: exists j, 0<=j<cellname.size()-1,
  //        cellname.substr(0,j) is all alphabetic (except for a
  //              possible cellname[0]=='$')
  //         && cellname.substr(j) is all numeric (except for a
  //              possible cellname[j]=='$') with at least one non-zero
  //              digit

  CellName (unsigned columnNumber = 0, unsigned rowNumber = 0,
            bool fixTheColumn = false,
            bool fixTheRow=false);

  std::string toString() const;
  // render the entire CellName as a string

  // Get components in spreadsheet notation
  std::string getColumn() const;
  int getRow() const;

  bool isRowFixed() const;
  bool isColumnFixed() const;


  // Get components as integer indices in range 0..
  int getColumnNumber() const;
  int getRowNumber() const;


  bool operator== (const CellName& r) const
    <:ensuremath{vdots}:>
private:
    <:ensuremath{vdots}:>
```



Arguably, the diagram on the left presents much the same information.

. . . .

Example: a book

If we were to try to capture our book abstraction (concentrating on the metadata), we might come up with something like:

```
class Book {
public:
  Book (Author)                 // for books with single authors
  Book (Author[], int nAuthors) // for books with multiple authors

  std::string getTitle() const;
  void putTitle(std::string theTitle);

  int getNumberOfAuthors() const;

  std::string getIsBN() const;
  void putISBN(std::string id);

  Publisher getPublisher() const;
  void putPublisher(const Publisher& publ);

  AuthorPosition begin();
  AuthorPosition end();
  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:
  <:ensuremath{vdots}:>
};
```

- What are `Author` and `Publisher` in this interface?
  - They are simply other ADTs in this library world, and will need to have designed interfaces of their own.

. . . .

Example: positions within a container

Example: positions within a container

Coming up with a good interface for our position abstraction is a problem that has challenged many an ADT designer.

- A look at our `Book` interface may suggest why.

```
class Book {
public:
  Book (Author)                  // for books with single authors
  Book (Author[], int nAuthors) // for books with multiple authors

  std::string getTitle() const;
  void putTitle(std::string theTitle);

  int getNumberOfAuthors() const;

  std::string getIsBN() const;
  void putISBN(std::string id);

  Publisher getPublisher() const;
  void putPublisher(const Publisher& publ);

  typedef int AuthorPosition;
  <+1>Author getAuthor (AuthorPosition authorNum) const;  <-1>

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:
  ⋮
};
```

- One intuitive idea might be to simply number the authors and treat the number as a position indicator, as shown here.

....

A problem with this is that the `getAuthor` function could then be done efficiently only if the authors inside each book were stored in an array or array-like data structure. And then `addAuthor` and `removeAuthor` cannot be implemented efficiently. Arrays also pose a difficulty – how large an array should be allocated for this purpose? If we allocate too few, the program crashes. So programmers usually wind up allocating an array large enough to contain as many items as possible – in this case as many authors as have ever collaborated on a single book. This means a lot of wasted storage for most books.

Both C++ and Java provide “expandable” array types, `std::vector` and `java.util.ArrayList`, respectively, that grow to accommodate however much data you actually insert into them. These resolve the storage issue, at a slight cost in speed, but still do not permit efficient implementation of `addAuthor` and `removeAuthor`.

Iterators

The solution adapted by the C++ community is to have every ADT that is a “container” of sequences of other data to provide a special type for positions within that sequence.

- The container itself provides functions to return

  - the beginning position in the sequence and
  - the position just *after* the last data item in the
    (these are typically called `begin()` and `end()`)
- The position ADT must provide, at a minimum:
  - A function to fetch the data item at the given position.
  - A function to advance from the current position to the next position in the sequence.
  - A function to compare two positions to see if they are the same.

....

A Possible Position Interface

In theory, we could satisfy this requirement with an ADT like this:

```
class AuthorPosition {
public:
   AuthorPosition();

   // get data at this position
   Author getData() const;

   // get the position just after this one
   AuthorPosition next() const;

   // Is this the same position as pos?
   bool operator== (const AuthorPosition& pos) const;
   bool operator!= (const AuthorPosition& pos) const;

};
```

which in turn would allow us to access authors like this:

```
void listAllAuthors(Book& b)
{
   for (AuthorPosition p = b.begin(); p != b.end();
        p = p.next())
     cout << "author: " << p.getData() << endl;
}
```

. . . .

The Iterator ADT

For historical reasons (and brevity), however, C++ programmers use overloaded operators for the `getData()` and `next()` operations:

```
class AuthorPosition {
public:
   AuthorPosition();

   // get data at this position
   Author operator*() const;

   // get a data/function member at this position
   Author* operator->() const;

   // move forward to the position just after this one
   AuthorPosition operator++();

   // Is this the same position as pos?
   bool operator== (const AuthorPosition& pos) const;
   bool operator!= (const AuthorPosition& pos) const;
};
```

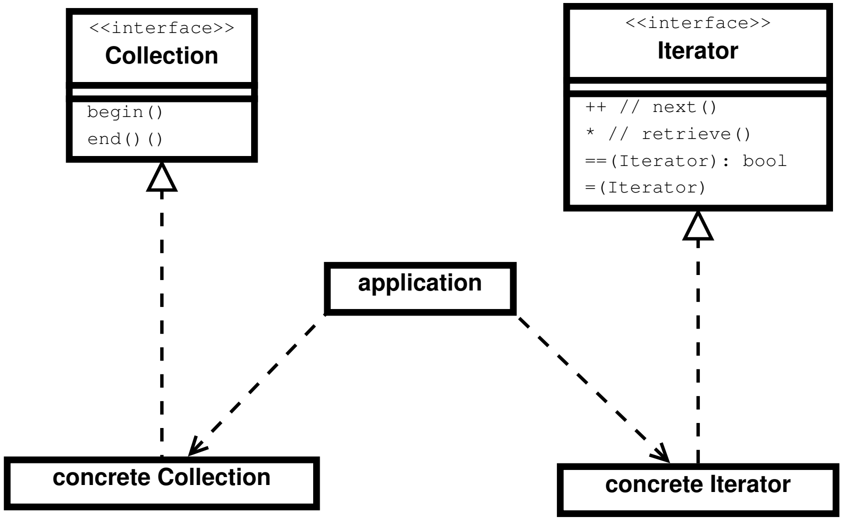
so that code to access authors would look like this:

```
void listAllAuthors(Book& b)
{
   for (AuthorPosition p = b.begin(); p != b.end();
        ++p)
     cout << "author: " << *p << endl;
}
```

This ADT for positions is called an *iterator* (because it lets us iterate over a collection of data).
Java has similar ADTs for positions within a container, called `Enumeration` and `Iterator`, which we will see in a later lesson.

3.2.2 Design Patterns

Iterator as a Design Pattern



The idea of an iterator is an instance of what we call a *design pattern*:

- a reusable concept that is not so much a piece of code as it is a design idea.

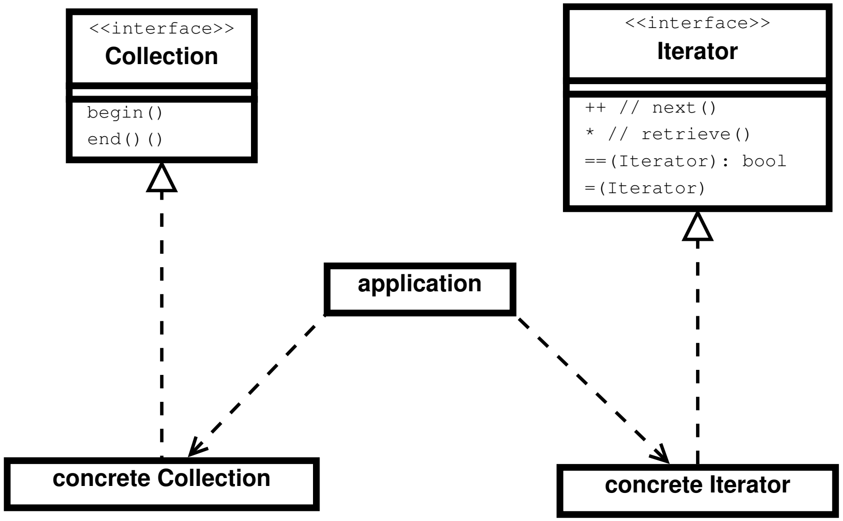
Pattern, not ADT

In C++, our application code does not actually work with an actual ADT named "Iterator".

- Instead, we typically have a lot of different ADTs, all of which share the common *pattern* of supporting an operator ++ to move forward, an operator $*$ for fetching a value at the position, etc.
- Each of these iterator ADTs is related to some kind of collection of data.
  - These collections have nothing to do with one another except that they share the common idea of supplying `begin()` and `end()` functions to provide the beginning and ending position within that collection. Again, there's probably no class in our application code actually named "Collection".

. . . .

Realizing a Design Pattern



- *Iterator* and *Collection* are general patterns for interfaces.
- We will have many actual classes that are unrelated to one another but that *implement* or *realize* those patterns.
  - Such classes are called *concrete* realizations of the general patterns.
  - It's these concrete classes that our application actually works with.

. . . .

You may have noticed that your textbook is titled "Objected-Oriented Design *& Patterns*". Keep an eye out, as we move through the semester, for more instances of common design patterns. (You might want to compare this diagram to the more Java-oriented version of this pattern on page 178 - in Java there really *is* something in the library named Iterator and our application works directly with that and only indirectly with the concrete realization.)

3.3 ADTs as contracts

ADTs as contracts

An ADT represents a contract between the ADT developer and the users (application programmers).

The Contract

- Application writers[a] are expected to alter/examine values of this type only via the operations and members provided.
- The creator of the ADT promises to leave the operation specifications unchanged.
- The creator of the ADT is allowed to change the code of the operations at any time, as long as it continues to satisfy the specifications.
- The creator of the ADT is also allowed to change the data structure actually used to implement the type.

[a] the "users" of the ADT

. . . .

Why the Contract?

What do we gain by holding ourselves to this contract?

- Application programmers can be designing and even implementing the application before the details of the ADT implementation have been worked out. This helps in
  - top-down design
  - development by teams
- The ADT implementors knows exactly what they must provide and what they are allowed to change.
- ADTs designed in this manner are often re-usable. By reusing code, we save time in
  - implementation
  - debugging
- We gain the flexibility to try/substitute different data structures to actually implement the ADT, *without needing to alter the application code.*
- By encouraging modularity, application code becomes more readable.

. . . .

Look back at the sample ADTs from the previous sections. Note that, although none of them contain data structures or algorithms to actually provide the required functions, all of them provide enough information that you could start writing application code using them.

3.3.1 Information Hiding

Information Hiding

Every design can be viewed as a collection of "design decisions".

- David Parnas formulated the principle: "Every module [procedure] should be designed so as to hide one design decision from the rest of the program."

  - He argued that such information hiding made future changes more economical.

. . . .

Encapsulation

Although ADTs can be designed without language support, they rely on programmers' self-discipline for enforcement of information hiding.

- Like "airline food" and "military intelligence", some would argue that "programmers' self-discipline" is an oxymoron.
- So programming languages evolved to provide enforcement of the ADT contract.

Encapsulation is the enforcement of information hiding by programming language constructs. In C++, this is accomplished by allowing ADT implementors to put some declarations into a `private:` area. Any application code attempting to use those private names will fail to compile.

. . . .

3.4 ADT Implementations

ADT Implementations

An ADT is *implemented* by supplying

- a *data structure* for the type name.
- coded *algorithms* for the operations.

We sometimes refer to the ADT itself as the *ADT specification* or the *ADT interface*, to distinguish it from the code of the *ADT implementation*.

In C++, implementation is generally done using a C++ `class`.

- Uses public/private to enforce the ADT contract

. . . .

3.4.1 Examples

Calendar Day Implementations

Read section 3.3 of your text (Horstmann) for a discussion of three different implementations of the `Day` ADT. notice how we can choose among implementations for performance reasons, without breaking any application code that relies on the `Day` interface.

. . . .

CellName implementation

```
class CellName
{
public:
  CellName (std::string column, int row,
            bool fixTheColumn = false,
            bool fixTheRow=false);
  //pre: column.size() > 0 && all characters in column are alphabetic
  //      row > 0

  CellName (std::string cellname);
  //pre: exists j, 0<=j<cellname.size()-1,
  //        cellname.substr(0,j) is all alphabetic (except for a
  //             possible cellname[0]=='$')
  //        && cellname.substr(j) is all numeric (except for a
  //             possible cellname[j]=='$') with at least one non-zero
  //             digit

  CellName (unsigned columnNumber = 0, unsigned rowNumber = 0,
            bool fixTheColumn = false,
            bool fixTheRow=false);
  std::string toString() const;
  // render the entire CellName as a string

  // Get components in spreadsheet notation
  std::string getColumn() const;
  int getRow() const;

  bool isRowFixed() const;
  bool isColumnFixed() const;


  // Get components as integer indices in range 0..
  int getColumnNumber() const;
  int getRowNumber() const;


  bool operator== (const CellName& r) const
    {return (columnNumber == r.columnNumber &&
             rowNumber == r.rowNumber &&
             theColIsFixed == r.theColIsFixed &&
             theRowIsFixed == r.theRowIsFixed);}

private:
  ⋮
  int rowNumber;
  bool theRowIsFixed;
  bool theColIsFixed;

  int CellName::alphaToInt (std::string columnIndicator) const;
  std::string CellName::intToAlpha (int columnIndex) const;
```

```
};


inline
bool CellName::isRowFixed() const {return theRowIsFixed;}

inline
bool CellName::isColumnFixed() const {return theColIsFixed;}



#endif
```

There are some options here the have not been explored:

- Do we want the column info stored as a number, a string, or both?

. . . .

Book implementation

We can implement Book in `book.h`:

```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                          // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors



  std::string getTitle() const;
  void setTitle(std::string theTitle);

  int getNumberOfAuthors() const;

  std::string getISBN() const;
  void setISBN(std::string id);

  Publisher getPublisher() const;
  void setPublisher(const Publisher& publ);

  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  static const int MAXAUTHORS = 12;
  Author authors[MAXAUTHORS];
```

```
};

#endif
```

and in `book.cpp`:

```
#include "book1.h"

  // for books with single authors
Book::Book (Author a)
{
  numAuthors = 1;
  authors[0] = a;
}

// for books with multiple authors
Book::Book (const Author au[], int nAuthors)
{
  numAuthors = nAuthors;
  for (int i = 0; i < nAuthors; ++i)
    {
      authors[i] = au[i];
    }
}

std::string Book::getTitle() const
{
  return title;
}

void Book::setTitle(std::string theTitle)
{
  title = theTitle;
}

int Book::getNumberOfAuthors() const
{
  return numAuthors;
}

std::string Book::getISBN() const
{
  return isbn;
}

void Book::setISBN(std::string id)
{
  isbn = id;
}

Publisher Book::getPublisher() const
{
  return publisher;
}

void Book::setPublisher(const Publisher& publ)
{
  publisher = publ;
}

Book::AuthorPosition Book::begin() const
{
  return authors;
```

```
}

Book::AuthorPosition Book::end() const
{
  return authors+numAuthors;
}


void Book::addAuthor (Book::AuthorPosition at, const Author& author)
{
  int i = numAuthors;
  int atk = at - authors;
  while (i >= atk)
    {
      authors[i+1] = authors[i];
      i--;
    }
  authors[atk] = author;
  ++numAuthors;
}


void Book::removeAuthor (Book::AuthorPosition at)
{
  int atk = at - authors;
  while (atk + 1 < numAuthors)
    {
      authors[atk] = authors[atk + 1];
      ++atk;
    }
  --numAuthors;
```

```
}
```

We'll explore some of the details and alternatives of these implementations in the next lesson.

. . . .

Chapter 4

Implementing ADTs with C++ Classes

4.1 Implementing ADTs in C++

Implementing ADTs

An ADT is *implemented* by supplying

- a *data structure* for the type name.
- coded *algorithms* for the operations.

We sometimes refer to the ADT itself as the *ADT specification* or the *ADT interface*, to distinguish it from the code of the ADT implementation.

. . . .

In C++, this is generally done using a C++ `class`. The class enforces the ADT contract by dividing the information about the implementation into `public` and `private` portions. The ADT designer declares all items that the application programmer can use as public, and makes the rest private. An application programmer who then tries to use the private information will be issued error messages by the compiler.

4.1.1 Declaring New Data Types

Declaring New Data Types

In the course of designing a program, we discover that we need a new data type. What are our options?

- Use an existing type
- Build our ADT "from scratch" by declaring a new class
- Inherit from an existing class and make slight modifications

. . . .

Well, to a certain degree, it depends upon whether or not we already have a data type that provides the interface (data and functions) that we want for this new one.

Use an Existing Type

On rare occasions, we may be lucky enough to have an existing type that provides *exactly* the capabilities we want for our new type. In that case, we can get by just using a `typedef` to create an "alias" for the existing type. For example, suppose that we were writing a program to look up the zip code for any given city. Obviously, one of the kinds of data we will be manipulating will be "city names". Now, a city name is pretty much just an ordinary character string, and anything we can do to a character string we could probably also do to a city name. So we might very well decide to take advantage of the existing C++ `std::string` data type and declare our new one this way:

```
typedef std::string CityName;
```

`typedef` basically gives us a way to attach a more convenient name to an existing type.

. . . .

Build our ADT "from scratch"

- Specify exactly what data and functions we want to associate with the new ADT.
- We do this by declaring a new class, e.g.,

```
class Book {
public:
  Book (Author)                 // for books with single authors
  Book (Author[], int nAuthors) // for books with multiple authors

  std::string getTitle() const;
  void putTitle(std::string theTitle);

  int getNumberOfAuthors() const;

  std::string getIsBN() const;
  void putISBN(std::string id);

  Publisher getPublisher() const;
  void putPublisher(const Publisher& publ);

  AuthorPosition begin();
  AuthorPosition end();

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:
  ⋮
};
```

. . . .

We may need to build our ADT “from scratch”, specifying exactly what data and functions we want to associate with the new ADT. This is done by declaring a new class and listing the data and function members we need to achieve our desired abstraction. For example, in building a system to track the inventory of a library, we wanted a Book ADT. In the prior lesson, we saw how we might declare an ADT for this purpose.

Inherit from an existing class

In between those two extremes, we sometimes have an existing type that provides almost everything we want for our new type, but needs just a little more data or a few more functions to make it what we want.

- Create an extended version of the existing type using inheritance

```
class BookInSeries: public Book {
   public:
     std::string getSeriesTitle() const;
     void putSeriesTitle(std::string theSeries);

     int getVolume() const;
     void putVolume(int);
   private:
     std::string seriesTitle;
     int volume;
};
```

. . . .

In that case we may be able to create an extended version of the existing type by using inheritance. For example, suppose that our library has a number of books that are grouped into series (e.g., “Cookbooks of the World”, volumes 1-28). This leads to the idea (abstraction) of a "book in series", that would be identical to a regular `Book` except for providing two addition data items, a series title and a volume number. We might then write the declaration shown here, which creates a new type named BookInSeries“ ” that is identical to the existing type Book except that it carries two additional data fields and the functions that provide access to them.

Inheritance is a powerful technique that lies at the heart of object-oriented programming. We’ll explore it in much more detail in later lessons.

4.1.2 Data Members

Data Members

```
class Book {
 public:
  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  static const int maxAuthors = 12;
  Author* authors;  // array of Authors
};
```

- To be useful, an ADT must usually contain some internal data. These are declared as *data members* of the class.

. . . .

To continue our prior example, what data should we associate with a book? Earlier, we said that a `Book` has a title, one or more authors, a publisher, and a unique ISBN. We can declare these as shown here. Note that it's not unusual for data members to involve still other ADTs (e.g., the `Author` data type in this example is itself presumably a class with a number of different data fields, including name, address, etc.).

Privacy

```
class Book {
 public:
    ⋮
 private:
  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  static const int maxAuthors = 12;
  Author* authors;  // array of Authors
};
```

....

In fact, we seldom declare ADTs with a large number of public data members as shown on the previous page. For a variety of reasons, we usually make data members *private*, meaning that they can only be accessed by code that is itself a part of the ADT.

Of course, if that's all we did, the ADT would be of little practical use. It would be rather like building a house with sturdy walls but no exterior doors or windows. If you can't get anything in and out, what use is it? We need some way to affect the data values and to examine them.

4.1.3 Function Members

Function Members

In addition to data, we can associate selected functions with our ADTs. For example, the missing access to the book data in our prior example can be provided by such *function members*.

```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                       // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const;
  void setTitle(std::string theTitle);

  int getNumberOfAuthors() const;
  std::string getISBN() const;
  void setISBN(std::string id);

  Publisher getPublisher() const;
  void setPublisher(const Publisher& publ);

  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  static const int MAXAUTHORS = 12;
  Author authors[MAXAUTHORS];

};

#endif
```

- Now, any member functions that we declare must eventually be implemented as well by providing the appropriate bodies.

  The usual convention is to package each ADT into its own pair of appropriately named `.h` and `.cpp` files. The class declaration for `Book` would typically appear in a file named `book.h`, as shown above.

  Then, in a separate file named `book.cpp` we would place the function definitions (bodies).

```
#include "book1.h"

  // for books with single authors
Book::Book (Author a)
{
  numAuthors = 1;
  authors[0] = a;
}

// for books with multiple authors
Book::Book (const Author au[], int nAuthors)
{
  numAuthors = nAuthors;
  for (int i = 0; i < nAuthors; ++i)
    {
      authors[i] = au[i];
    }
}

std::string Book::getTitle() const
{
  return title;
}

void Book::setTitle(std::string theTitle)
{
  title = theTitle;
}

int Book::getNumberOfAuthors() const
{
  return numAuthors;
}

std::string Book::getISBN() const
{
  return isbn;
}

void Book::setISBN(std::string id)
{
  isbn = id;
}

Publisher Book::getPublisher() const
{
  return publisher;
}

void Book::setPublisher(const Publisher& publ)
{
  publisher = publ;
}

Book::AuthorPosition Book::begin() const
{
  return authors;
}

Book::AuthorPosition Book::end() const
{
  return authors+numAuthors;
```

```
}


void Book::addAuthor (Book::AuthorPosition at, const Author& author)
{
  int i = numAuthors;
  int atk = at - authors;
  while (i >= atk)
    {
      authors[i+1] = authors[i];
      i--;
    }
  authors[atk] = author;
  ++numAuthors;
}


void Book::removeAuthor (Book::AuthorPosition at)
{
  int atk = at - authors;
  while (atk + 1 < numAuthors)
    {
      authors[atk] = authors[atk + 1];
      ++atk;
    }
  --numAuthors;
}
```

ADTs need not be complicated

None of the functions in that ADT are particularly complex.

- It's a common mistake to believe that an abstract idea only "needs" to be an ADT if it is complicated.
- Instead, an abstraction should become and ADT if doing so makes the code that uses it more expressive.
  - Most functions in a typical ADT are basic get/set *attribute* functions.

. . . .

Inline Functions

Many of the member functions in this example are simple enough that we might consider an alternate approach, declaring them as *inline functions*, in which case the `book.h` file would look something like this:

```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                         // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }
  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  static const int MAXAUTHORS = 12;
  Author authors[MAXAUTHORS];

};

#endif
```

and the book.cpp file would be reduced to this:

```
#include "book1.h"
```

```
  // for books with single authors
Book::Book (Author a)
{
  numAuthors = 1;
  authors[0] = a;
}

// for books with multiple authors
Book::Book (const Author au[], int nAuthors)
{
  numAuthors = nAuthors;
  for (int i = 0; i < nAuthors; ++i)
    {
      authors[i] = au[i];
    }
}


Book::AuthorPosition Book::begin() const
{
  return authors;
}

Book::AuthorPosition Book::end() const
{
  return authors+numAuthors;
}


void Book::addAuthor (Book::AuthorPosition at, const Author& author)
{
  int i = numAuthors;
  int atk = at - authors;
  while (i >= atk)
    {
      authors[i+1] = authors[i];
      i--;
    }
  authors[atk] = author;
  ++numAuthors;
}


void Book::removeAuthor (Book::AuthorPosition at)
{
  int atk = at - authors;
  while (atk + 1 < numAuthors)
    {
      authors[atk] = authors[atk + 1];
      ++atk;
    }
  --numAuthors;
}
```

. . . .

Before leaving this example, let's fill out the declarations for the other key ADTs in our library example, starting with the `Author`...

```
class Author
{
public:
  std::string getName() const          {return name;}
  void putName (std::string theName) {name = theName;}

  const Address& getAddress() const    {return address;}
  void putAddress (const Address& addr) {address = addr;}

  long getIdentifier() const      {return identifier;}

private:
  std::string name;
  Address address;
  const long identifier;
};
```

...and then the `Address`.

```
class Address {
public:
  std::string getStreet() const;
  void putStreet (std::string theStreet);

  std::string getCity() const;
  void putCity (std::string theCity);

  std::string getState() const;
  void putState (std::string theState);

  std::string getZip() const;
  void putZip (std::string theZip);

private:
  std::string street;
  std::string city;
  std::string state;
  std::string zip;
};
```

Most of the member functions in these examples are awfully simple. That's not unusual. The majority of ADT member functions really are very simple. There's nothing wrong with that. ADTs do not have to be complicated to be useful.

4.2 Special Functions

4.2.1 Constructors

Initializing Data

```
class Address {
public:
  std::string getStreet() const;
  void putStreet (std::string theStreet);

  std::string getCity() const;
  void putCity (std::string theCity);

  std::string getState() const;
  void putState (std::string theState);

  std::string getZip() const;
  void putZip (std::string theZip);

private:
  std::string street;
  std::string city;
  std::string state;
  std::string zip;
};
```

```
class Author
{
public:
  std::string getName() const        {return name;}
  void putName (std::string theName) {name = theName;}

  const Address& getAddress() const   {return address;}
  void putAddress (const Address& addr) {address = addr;}

  long getIdentifier() const     {return identifier;}

private:
  std::string name;
  Address address;
  const long identifier;
};
```

- One of the weaknesses of the ADT design shown above is that there is no easy way to initialize new address and author objects.
- Of course, we could do it one data field at a time:

```
Address addr;
addr.putStreet ("21 Pennsylvania Ave.");
addr.putCity ("Washington");
addr.putState ("D.C.");
addr.putZip ("10001");

Author doe;
doe.putName ("Doe, John");
doe.putAddress (addr);
```

. . . .

Problems with Field-By-Field Initialization

- It would be quite easy to forget to initialize one or more of the data fields.
- As the code gets modified over the course of the project, some misguided programmer might place some additional lines of code in between these. (This is particularly likely if we need a nontrivial computation to come up with the values for these data fields.)

  That leads to a real possibility of our using `addr` before all the data fields have been initialized.
- There's no way to initialize the Author's `identifier` data.
  - We don't have a `putIdentifier` function because we did not want to allow arbitrary changes to that data.
  - Note that the `Author::identifier` field is declared as `const`.
- Last, but not least, the process just takes too long - too many lines of code written just to initialize one data object.

. . . .

Constructor Functions

C++ provides a special kind of member function to streamline the initialization process. It's called a *constructor*.

Constructors are functions. They can be called just like any other function, can be declared to take any parameters we think they need to get their job done, and they do return a result value, just like ordinary functions. What makes constructors unusual is that their name must be the same as their "return type", the name of the class being initialized. A constructor is called when we define a new variable, and any parameters supplied in the definition are passed as parameters to the constructor.

Suppose we add a constructor to each of our *Address* and *Author* classes.

```
class Address {
public:
  Address (std::string theStreet, std::string theCity,
           std::string theState, std::string theZip);
  std::string getStreet() const;
  void putStreet (std::string theStreet);

  std::string getCity() const;
  void putCity (std::string theCity);

  std::string getState() const;
  void putState (std::string theState);

  std::string getZip() const;
  void putZip (std::string theZip);

private:
  std::string street;
  std::string city;
  std::string state;
  std::string zip;
};


class Author
{
public:
  Author (std::string theName, Address theAddress, long id);

  std::string getName() const        {return name;}
  void putName (std::string theName) {name = theName;}

  const Address& getAddress() const   {return address;}
  void putAddress (const Address& addr) {address = addr;}
  long getIdentifier() const     {return identifier;}

private:
  std::string name;
  Address address;
  const long identifier;
};
```

. . . .

Declaring Variables with Constructors

Then we can create a new author object much more easily:

```
Address addr ("21 Pennsylvania Ave.",
              "Washington",
              "D.C.", "10001");

Author doe ("Doe, John", addr, 1230157);
```

or, since the `addr` variable is probably only there only for the purpose of initializing this author and is probably not used elsewhere, we can do:

```
Author doe ("Doe, John",
            Address (
             "21 Pennsylvania Ave.",
             "Washington",
             "D.C.", "10001"),
             1230157);
```

. . . .

Implementing Constructors

The implementation of this constructor is pretty straightforward.

```
Address::Address
  (std::string theStreet, std::string theCity,
   std::string theState, std::string theZip)
{
  street = theStreet;
  city = theCity;
  state = theState;
  zip = theZip;
}
```

. . . .

Initialization Lists

The implementation of the `Author` constructor has one complication.

```
Author::Author (std::string theName,
                Address theAddress, long id)
  : identifier (id)
{
  name = theName;
  address = theAddress;
}
```

- We *can't* say

  ```
  identifier = id;
  ```

  in the function body, because `identifier` was declared as being `const` and so cannot be assigned to.

. . . .

C++ provides a special syntax for initializing constant data members. It's called an *initialization list* and is shown in the highlighted portion of the constructor code.

Initialization lists appear only in constructors and *must* used for constants, references (which also cannot be assigned to) and to initialize data members that need elaborate constructor calls of their own. They *can* be used to initialize any data member, so an alternate implementation of this constructor would be:

Example: Alternate Constructors

```
Author::Author (std::string theName,
                Address theAddress, long id)
  : identifier (id)
{
  name = theName;
  address = theAddress;
}
```

or

```
Author::Author (std::string theName,
                Address theAddress, long id)
   : name(theName), address(theAddress),
    identifier(id)
{
}
```

The second constructor might run slightly faster.

. . . .

In the first constructor, the *address* will first be initialized using `Address()`, then in the body we copy over the freshly initialized value when we assign *theAddress* to it. That means that the tie spent initialliy initializing the data member is probably wasted.

Book Constructors

```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                         // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;
  static const int MAXAUTHORS = 12;
  Author authors[MAXAUTHORS];

};

#endif
```

Our Book class needs constructors as well, but is slightly complicated by the fact that books can have multiple authors. How can we pass an arbitrary number of authors to a constructor (or to any function, for that matter)?

In this case, we'll do it by passing an array of Authors together with an integer indicating how many items are in the array.

To implement the constructors, we add each author, however many we are given, into the data structure we chose to hold Authors within our Book ADT. In this case, we use a simple array.

```
// for books with multiple authors
Book::Book (const Author au[], int nAuthors)
{
  numAuthors = nAuthors;
  for (int i = 0; i < nAuthors; ++i)
    {
      authors[i] = au[i];
    }
}
```

With that constructor in place, we could initialize books by first building an appropriate author array, then declaring our new Book object. For example, given these authors:

```
Author budd
   ("Budd, Timothy",
    Address("21 Nowhere Dr.", "Podunk", "NY", "01010"),
    1123);
Author doe
   ("Doe, John",
    Address("212 Baker St.", "Peoria", "IL", "12345"),
    1124);
```

```
Author smith
   ("Smith, Jane",
    Address("47 Scenic Ct.", "Oahu", "HA", "54321"),
    1125);
```

we can create some books like this:

```
Author textAuthors[] = {budd};
Book text361 ("Data Structures in C++", 1,
              textAuthors, "0-201-10758");
Book ootext ("Introduction to Object Oriented Programming",
             1, textAuthors, "0-201-12967");

Author recipeAuthors[] = {doe, smith};
Book recipes ("Cooking with Gas", 2, recipeAuthors, "0-124-46821");
```

In cases were we really do have multiple authors (e.g., `recipes`), that's probably as simple as it's going to get. It's a bit awkward for books that only have a single author, however. But, just as with any other functions in C++, we can *overload* functions - we can have any number of functions of the same name as long as the formal parameter list for each constructor is different. Since single authorship is likely to be a common case, it might be convenient to declare another `Book` constructor to serve that special case.

```
class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author)                  // for books with single authors
  Book (Author[], int nAuthors) // for books with multiple authors

  std::string getTitle() const         { return title; }
    ⋮
```

We could then use either constructor, as appropriate, when creating `Book`s.

```
Book text361 ("Data Structures in C++",
              budd, "0-201-10758");
Book ootext ("Introduction to Object Oriented Programming",
             budd, "0-201-12967");

Author* recipeAuthors[] = {doe, smith};
Book recipes ("Cooking with Gas", 2, recipeAuthors, "0-124-46821");
```

4.2.2 Destructors

Destructors

The flip-side of initializing new objects is cleaning up when old objects are going away. Just as C++ provides special functions, constructors, for handling initialization, it also provides special functions, destructors, for handling clean-up.

A *destructor*for the class `Foo` is a function of the form

```
~Foo();
```

Destructors are never called explicitly. Instead the compiler generates a call to an object's destructor for us.

. . . .

The Compiler Calls a Destructor when...

- Execution passes outside of the `{...}` within which the object was declared. For example, if we wrote

  ```
  if (someTest)
    {
     Book b = text361;
     cout << b.getTitle() << endl;
    }
  ```

  what the compiler would actually generate would be something along the lines of

  ```
  if (someTest)
    {
  ```

```
   Book b = text361;
   cout << b.getTitle() << endl;
   b.~Book();  // implicitly generated by the compiler
  }
```

. . . .

The Compiler Calls a Destructor when...

- When we `delete` a pointer to an object, the object's destructor is (implicitly) called prior to actually recovering the memory occupied by the object. For example, if we were to write:

```
Book *bPointer = new Book(text361); // initialized using copy constructor
   ⋮
cout << bPointer->getTitle() << endl;
delete bPointer;
```

  what the compiler would actually generate would be something along the lines of

```
Book *bPointer = new Book(text361); // initialized using copy constructor
    ⋮
cout << bPointer->getTitle() << endl;
bPointer->~Book();
free(bPointer);  // recover memory at the address in bPointer
```

. . . .

Inside a Destructor

The most common uses for destructors is to clean up allocated memory for an object that is about to disappear.

- If we have used new to allocate any memory on the heap for one or more data members of the object,

  - we usually `delete` that memory in the destructor.

. . . .

4.2.3 Operators

Operators

One of the truly delightful, or demonic, depending on your point of view, aspects of C++ is that: Almost every operator in the language can be declared in our own classes.

In most programming languages, we can write things like " `x+y` " or " `x<y` " only if `x` and `y` are integers, floating point numbers, or some other predefined type in the language.In C++, we can add these operators to any class we design, if we feel that they are appropriate.

The basic idea is really very simple. Almost all of the things we use as operators,

- including `+ - * / | & < > <= >= = == != ++ -- -> += -= *= /=`,
- and some things you probably don't consider as operators, such as the `[ ]` used in array indexing and the `( )` used in function calls,
- but not `.` or `::`,

are actually "syntactic sugar" for a function whose name if formed by appending the operator itself to the word "`operator`".

. . . .

Operators as Shorthand

For example,

- `a + b*(-c)` is actually just a shorthand for

  ```
  operator+(a, operator*(b, operator-(c)))
  ```

- and if you write

  ```
  testValue = (x <= y);
  ```

is a shorthand for

```
operator=(testValue, operator<=(x, y);
```

. . . .

Declaring Operators

Now, this shorthand is so much easier to deal with that there's seldom any good reason to actually write out the long form of these calls. But knowing the shorthand means that we can now declare new operators in the same way we declare new functions. For example, we could declare a + operator for `Books` this way:

```
Book operator+ (const Book& left, const Book& right);
```

and then call it like this:

```
Book b = book1 + book2;
```

but that's probably not a good idea, because it's not clear just what it means to add two books together. So any implementation we wrote for the `operator+` function for `Books` would probably be nonintuitive and confusing to other programmers. There are, however, a few operators that *would* make sense for `Book` and for almost all ADTs.

. . . .

Assignment

Chief among these is the assignment operator.

- When we write `book1 = book2`, that's shorthand for `book1.operator=(book2)`

Assignment is used so pervasively that a class without assignment would be severely limited (although sometimes designers may *want* that limitation). Consequently, if a class does not provide an explicit assignment operator, the compiler will attempt to generate one. The compiler-generated version will simply assign each data member in turn.

We'll look more closely at when and how to write our own assignment operators in the next lesson.

. . . .

I/O

Another, **very** common set of operators that programmers often write for their own code are the I/O operators << and >>, particularly the output operator <<. Indeed, I would argue that you should always provide an output operator for every class you write, even if you don't expect to use it in your final application.

My reason for this is quite practical. Sooner or later, you're going to discover that your program isn't working right. So how are you going to debug it? The most common thing to do is to add debugging output at all the spots where you think things might be going wrong. So you come to some statement:

```
doSomethingTo(book1);
```

and you realize that it might be really useful to know just what the value of that book was before and after the call:

```
cerr << "Before doSomethingTo: " << book1 << endl;
doSomethingTo(book1);
cerr << "After doSomethingTo: " << book1 << endl;
```

Now, if you have already written that `operator<<` function for your `Book` class, you can proceed immediately. If you haven't written it already, do you really want to be writing and debugging that new function *now*, when you are already dealing with a different bug?

. . . .

Output Operator Example

Here's a possible output routine for our `Book` class. It simply prints the book's identifier on one line, prints the title on the next line, then prints all the authors separated by commas.

```
ostream& operator<< (ostream& out, const Book& b)
{
  out << b.getISBN() << "n";
  out << b.getTitle() << "n";
  for (AuthorPosition current = b.begin(); current != b.end(); ++current)
  {
    Author au = *current;
    out << au << ", ";;
  }
  out << "n  published by" << b.getPublisher();
  out << endl;
  return out;
}
```

- This assumes that we have implemented an output operator for *Author*.
  - But if you bought into the idea of providing an output operator for every class you write, that one should already exist.

. . . .

Output Operator Example

```
ostream& operator<< (ostream& out, const Book& b)
{
  out << b.getISBN() << "n";
  out << b.getTitle() << "n";
  for (AuthorPosition current = b.begin(); current != b.end(); ++current)
  {
    Author au = *current;
    out << au << ", ";;
  }
  out << "n  published by" << b.getPublisher();
  out << endl;
  return out;
}
```

- The return statement at the end returns the output stream to which we are writing. It's the fact that << is an expression (with a returned value as its result) and that it returns the very stream we are writing to that let's us write "chains" of output expressions like:

```
cout << book1 << " is better than "
     << book2 << endl;
```

which is treated by the compiler as if we had written:

```
(((cout << book1) << " is better than ")
         << book2) << endl;
```

or, if you prefer,

```
operator<<(
           operator<<(
                      operator<<(
                                 operator<<(cout, book1),
                                 " is better than "
                                ),
                      book2
                     ),
           endl
          );
```

so that each output operation, in turn, passes the stream on to the next one in line.

....

Comparisons

After assignments and I/O, the most commonly programmed operators would be the relational operators, especially == and <. The compiler never generates these implicitly, so if we want them, we have to supply them.

```
class Address
{
   ⋮
  bool operator== (const Address&) const;
   ⋮
};
```

The trickiest thing about providing these operators is making sure we understand just what they should *mean* for each individual ADT. For example, if I were to write

```
if (address1 == address2)
```

what would I expect to be true of two `Addresses` that passed this test? Probably that the two addresses would have the same street, city, state, and zip - in other words, that all the data fields should themselves be equal. In that case, …

. . . .

Equality via All Data Members Equal

This would be a reasonable implementation:

```
bool Address::operator== (const Address& right) const
{
  return (street == right.street)
      && (city   == right.city)
      && (state  == right.state)
      && (zip    == right.zip);
}
```

We could do something similar for `Author`:

```
bool Author::operator== (const Author& right) const
{
  return (name       == right.name)
      && (address    == right.address)
      && (identifier == right.identifier);
}
```

(which, interestingly, makes use of the `Address::operator==` that we have just defined).

. . . .

Equality via "keys"

But there's actually another choice that might be reasonable. If every author has a unique, unchanging identifier, we might be able to just say:

```
bool Author::operator== (const Author& right) const
{
  return (identifier == right.identifier);
}
```

on the grounds that two `Author` objects with the same identifier value must actually be describing the same person, even if they are inconsistent in the other fields.

- That's a different *meaning* for ==.
  - Neither of these two possible meanings is obviously better or "more correct" than the other.
- In a real project, we would need to look hard at how we will use these objects to decide what meaning is appropriate for ==.

....

How Many Relational Ops do we Need?

- In C++ we usually provide only operators == and <
  - Many `std::` functions use these two, but none rely on the other 4 (`!= > >= <=`).
- The standard library contains functions defining each of these 4 additional relational operators in terms of == and <.
  - We get these additional operators by simply saying

    ```
    using namespace std::rel_ops;
    ```

....

We might or might not choose to define an `operator!=`. If we do, it's pretty simple, as shown here.

```
class Address
{
   <t>\smvdots</t>
  bool operator== (const Address&) const;

  /*+1*/bool operator!= (const Address& a) const {return !operator==(a);}  /*-1*/
   ⋮
};
```

On the other hand, if we don't provide this operator, it's easy enough for programmers using our classes to write tests like:

```
if (!(address1 == address2))
```

Similarly, if we provide any one of the remaining relational operators (< > <= >=) then we can derive the others from a combination of the one we provide and `operator==` and `!`. For example, if we have == and <, we can define the others in term of those two:

```
bool operator> (const WordSet& left, const WordSet& right)
{
  return (right < left);
}

bool operator>= (const WordSet& left, const WordSet& right)
{
  return !(right < left);
}
    ⋮
```

In fact, the standard library already contains functions defining each of these 4 additional relational operators in terms of the == and <, so we don't need to write these. We get the additional operators by simply saying

```
using namespace std::rel_ops;
```

Designing a Good <

In C++, we traditionally provide `operator<` whenever possible (and many code libraries assume that this operator is available).

Again, just what this should *mean* depends upon the abstraction we are trying to support, but there are a few hard and fast rules. We should always design `operator<` so that

- If `x < y` is true, then `y < x` and `x == y` must be false.
- If `x == y` is true, then `x < y` and `y < x` must be false.
- If `x == y` is false, then either `x < y` or `y < x` (but not both) must be true.

In other words, given any two values *x* and *y*, one and exactly one of the relations `x < y`, `y < x`, and `x == y` should be true.

. . . .

Example: Comparing Authors

For example, this would be a reasonable pair of comparison operators for `Authors`:

```
bool Author::operator== (const Author& right) const
{
  return (identifier == right.identifier);
}

bool Author::operator< (const Author& right) const
{
  return (identifier < right.identifier);
}
```

. . . .

Example: Comparing Addresses

Here is a reasonable pair for `Address`:

```
bool Address::operator== (const Address& right) const
{
  return (street == right.street)
      && (city   == right.city)
      && (state  == right.state)
      && (zip    == right.zip);
}

bool Address::operator< (const Address& right) const
{
  if (street < right.street)
    return true;
  else if (street == right.street)
    {
     if (city < right.city)
       return true;
     else if (city == right.city)
       {
        if (state < right.state)
          return true;
        else if (state == right.state)
          {
           if (zip < right.zip)
            return true;
          }
       }
    }
  return false;
}
```

. . . .

On the other hand, this pair does *not* work.

```
bool Address::operator== (const Address& right) const
```

```
{
  return (street == right.street)
      && (city   == right.city)
      && (state  == right.state)
      && (zip    == right.zip);
}

bool Address::operator< (const Address& right) const
{
  return (street < right.street)
      || (city   < right.city)
      || (state  < right.state)
      || (zip    < right.zip);
}
```

Can you explain why?

4.3 Example: Multiple Implementations of Book

Example: Multiple Implementations of Book

One of the characteristics we expect when working with ADTs is that the implementing data structures and algorithms can be altered without changing the interface.

. . . .

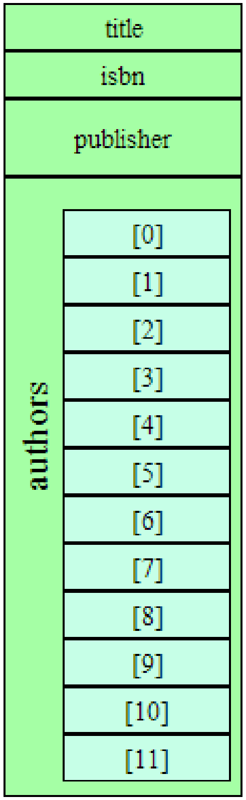
As an example of this, I will present 4 different implementations of the `Book` ADT. The first three should be well within the grasp of everyone in this course. The last will take advantage of a data structure familiar to most students who have taken a C++ -based data structures course, such as our own CS361.

4.3.1 Simple Arrays

The first of these is the one we have used so far in this lesson.

Simple Arrays

The authors are stored in an array of fixed size, statically allocated as a part of the Book object.



```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  static const int MAXAUTHORS = 12;
  Author authors[MAXAUTHORS];

};

#endif
```

....

In this approach, each book object can be illustrated as a single block of memory, within which can be found the individual data members such as title and ISBN, but also can be found a fixed number of slots for authors.

Adding and Removing

- The code to add and remove authors is rather typical array manipulation code.

```
#include "book1.h"

  // for books with single authors
Book::Book (Author a)
{
  numAuthors = 1;
  authors[0] = a;
```

```
}

// for books with multiple authors
Book::Book (const Author au[], int nAuthors)
{
  numAuthors = nAuthors;
  for (int i = 0; i < nAuthors; ++i)
    {
      authors[i] = au[i];
    }
}


Book::AuthorPosition Book::begin() const
{
  return authors;
}

Book::AuthorPosition Book::end() const
{
  return authors+numAuthors;
}


void Book::addAuthor (Book::AuthorPosition at, const Author& author)
{
  int i = numAuthors;
  int atk = at - authors;
  while (i >= atk)
    {
      authors[i+1] = authors[i];
      i--;
    }
  authors[atk] = author;
  ++numAuthors;
}


void Book::removeAuthor (Book::AuthorPosition at)
{
  int atk = at - authors;
  while (atk + 1 < numAuthors)
    {
      authors[atk] = authors[atk + 1];
      ++atk;
    }
  --numAuthors;
}
```

. . . .

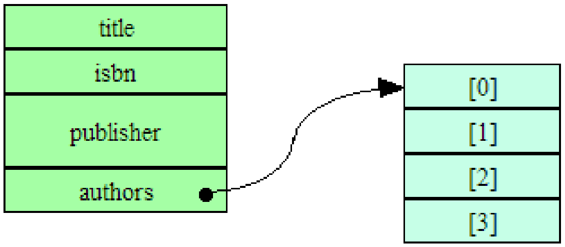
This approach has the advantage of simplicity - it's probably what most novice programmers would attempt. Its disadvantages include speed when adding or removing authors from very long lists (though, realistically, very few books are likely to have long enough lists of authors for this to really matter). More important, the choice of value for `MAXAUTHORS` is somewhat critical. Too small and we risk a program crash when a book is created with more authors than can fit in the array. Too large, and we waste a lot of storage for the vast majority of books that only have one or two authors.

4.3.2 Dynamically Allocated Arrays

Dynamically Allocated Arrays

A more flexible approach can be obtained by allocating the array of authors on the heap.

In this approach, each book object occupies two distinct blocks of memory, one of them an array allocated on the heap.



```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                          // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }


  AuthorPosition begin() const;
  AuthorPosition end() const;
  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  int MAXAUTHORS;
  Author* authors;

};

#endif
```

In the .h file, the statically sized array `authors` is replaced by a pointer to an (array of) `Author`.

. . . .

Dynamically Allocated Arrays (cont.)

```
#include "book2.h"

  // for books with single authors
Book::Book (Author a)
{
  MAXAUTHORS = 4;
  authors = new Author[MAXAUTHORS];
  numAuthors = 1;
  authors[0] = a;
}

// for books with multiple authors
Book::Book (const Author au[], int nAuthors)
{
  MAXAUTHORS = 4;
  authors = new Author[MAXAUTHORS];
  numAuthors = nAuthors;
  for (int i = 0; i < nAuthors; ++i)
    {
      authors[i] = au[i];
    }
}


Book::AuthorPosition Book::begin() const
{
  return authors;
}

Book::AuthorPosition Book::end() const
{
  return authors+numAuthors;
}


void Book::addAuthor (Book::AuthorPosition at, const Author& author)
{
  if (numAuthors >= MAXAUTHORS)
    {
      Author∗ newAuthors = new Author[2∗MAXAUTHORS];
      for (int i = 0; i < MAXAUTHORS; ++i)
    newAuthors[i] = authors[i];
      MAXAUTHORS ∗= 2;
      delete [] authors;
      authors = newAuthors;
    }
  int i = numAuthors;
  int atk = at - authors;
  while (i > atk)
    {
      authors[i] = authors[i-1];
      i--;
    }
  authors[atk] = author;
  ++numAuthors;
}


void Book::removeAuthor (Book::AuthorPosition at)
{
  int atk = at - authors;
  while (atk + 1 < numAuthors)
    {
      authors[atk] = authors[atk + 1];
      ++atk;
    }
  --numAuthors;
}
```

The code is still pretty straightforwardly array-based.

- The constructors are changed to now allocate the array on the heap.
- The array size allocated is smaller, because of a more sophisticated approach used in `addAuthor`.
  - There, before adding a new author to the book, we make a check to see if the array is already full.
  - If it is, we allocate a new, larger array, copy the former list of authors from the old array into the new one, discard the old array, and then proceed to add the new author into the new, larger array.

    That way we need not worry about choosing a fixed bound on the number of authors that can be accommodated in any book. (Some of you may recognize this expandable array approach as being typical of the `std::vector` class in the C++ standard library.)

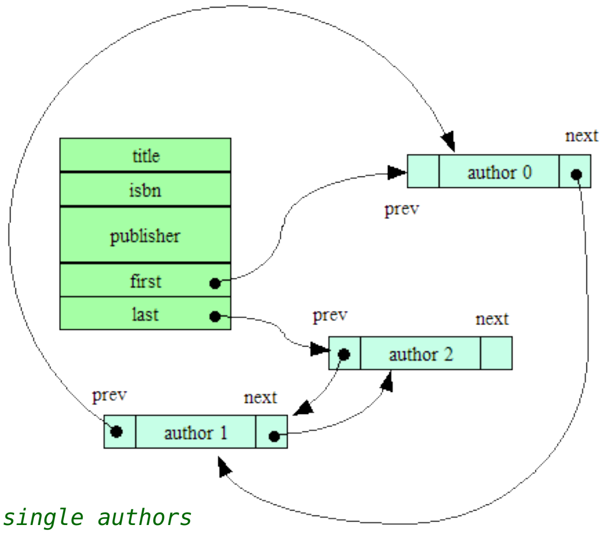
. . . .

4.3.3 Linked Lists

Linked Lists

A still more flexible approach can be obtained by using a linked list of authors. In this case, I have chosen a doubly-linked list (pointers going both forward and backward from each node).

In this approach, each book object occupies many blocks of memory, most of them being linked list nodes allocated on the heap.



```
#ifndef BOOK_H
#include "author.h"
#include "authoriterator.h"
#include "publisher.h"


class Book {
public:
  typedef AuthorIterator AuthorPosition;

  Book (Author);                          // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }
  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  AuthorNode* first;
  AuthorNode* last;

  friend class AuthorIterator;
};

#endif
```

The `Book` itself holds pointers to the first and the last node in the chain.

....

Linked Lists impl

```
#include "authoriterator.h"
#include "book3.h"

  // for books with single authors
```

```
Book::Book (Author a)
{
  numAuthors = 1;
  first = last = new AuthorNode (a, 0, 0);
}

// for books with multiple authors
Book::Book (const Author au[], int nAuthors)
{
  numAuthors = 0;
  first = last = 0;
  for (int i = 0; i < nAuthors; ++i)
    {
      addAuthor(end(), au[i]);
    }
}


Book::AuthorPosition Book::begin() const
{
  return AuthorPosition(first);
}

Book::AuthorPosition Book::end() const
{
  return AuthorPosition(0);
}


void Book::addAuthor (Book::AuthorPosition at, const Author& author)
{
  if (first == 0)
     first = last = new AuthorNode (author, 0, 0);
  else if (at.pos == 0)
    {
      last->next = new AuthorNode (author, last, 0);
      last = last->next;
    }
  else
    {
      AuthorNode* newNode = new AuthorNode(author, at.pos->prev, at.pos);
      at.pos->prev = newNode;
      if (at.pos == first)
    first = newNode;
      else
    newNode->prev->next = newNode;
    }
  ++numAuthors;
}


void Book::removeAuthor (Book::AuthorPosition at)
{
  if (at.pos == first)
    {
      if (first == last)
        first = last = 0;
      else
    {
      first = first->next;;
      first->prev = 0;
    }
    }
  else if (at.pos == last)
    {
      last = last->prev;
      last->next = 0;
    }
  else
    {
      AuthorNode* prv = at.pos->prev;
      AuthorNode* nxt = at.pos->next;
      prv->next = nxt;
      nxt->prev = prv;
    }
  delete at.pos;
  --numAuthors;
}
```

The code is considerably messier - pointer manipulation can be tricky, no matter how many times you do it.

- The code is quite efficient however.
- Adding and removing nodes remains quick no matter how many authors are actually in the list.

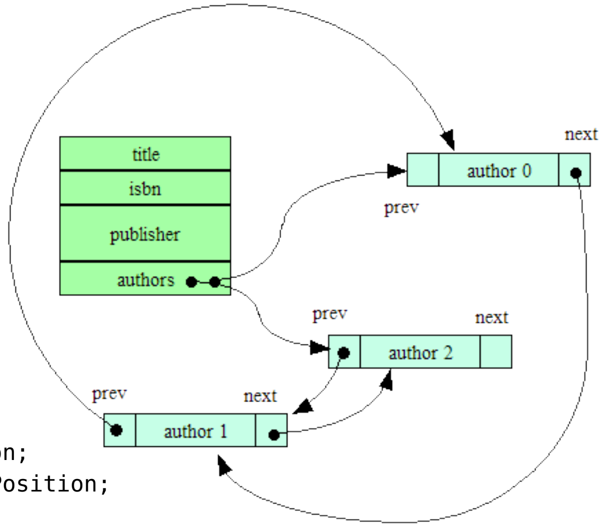
. . . .

4.3.4 Standard List

Those of you who have taken CS361 or a similar course might guess that we could save ourselves a lot of programming effort by exploiting one of the standard container data structured provided in the C++ std library. So, for our final implementation, I present an implementation based upon `std::list`.

Standard List

We presume that the data in this approach is arranged pretty much as before, though we don't really *know* that for sure because we trust the abstraction provided by the `std::list` ADT.



```
#ifndef BOOK_H

#include <list>

#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef std::list<Author>::const_iterator AuthorPosition;
  typedef std::list<Author>::const_iterator const_AuthorPosition;

  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  const_AuthorPosition begin() const;
  const_AuthorPosition end() const;

  AuthorPosition begin();
  AuthorPosition end();

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  std::list<Author> authors;
};

#endif
```

```
#include "authoriterator.h"
#include "book4.h"

  // for books with single authors
Book::Book (Author a)
{
  numAuthors = 1;
  authors.push_back(a);
```

```
}

// for books with multiple authors
Book::Book (const Author au[], int nAuthors)
{
  numAuthors = nAuthors;
  for (int i = 0; i < nAuthors; ++i)
    {
      authors.push_back(au[i]);
    }
}


Book::const_AuthorPosition Book::begin() const
{
  return authors.begin();
}

Book::const_AuthorPosition Book::end() const
{
  return authors.end();
}


Book::AuthorPosition Book::begin()
{
  return authors.begin();
}

Book::AuthorPosition Book::end()
{
  return authors.end();
}


void Book::addAuthor (Book::AuthorPosition at, const Author& author)
{
  authors.insert (at, author);
  ++numAuthors;
}


void Book::removeAuthor (Book::AuthorPosition at)
{
  authors.erase (at);
  --numAuthors;
}
```

The code is quite simple, at least for anyone already familiar with the `std::list` ADT.

. . . .

4.3.5 Implementing Book::AuthorPosition

Implementing Book::AuthorPosition

Our `Book` ADT calls for a "helper" ADT, the `AuthorPosition`, to represent the position of a particular author within the author list for the book.

. . . .

Although we have looked at the desired interface for iterators such as `AuthorPosition`, we skipped past its implementation while we were looking at implementing `Book`.

Iterator Interface Review

To review, our iterator must supply the following operations:

```
class AuthorPosition {
public:
   AuthorPosition();

   // get data at this position
   Author operator*() const;

   // get a data/function member at this position
   Author* operator->() const;

   // move forward to the position just after this one
   AuthorPosition operator++();

   // Is this the same position as pos?
   bool operator== (const AuthorPosition& pos) const;
   bool operator!= (const AuthorPosition& pos) const;

};
```

. . . .

A little thought will lead to the realization that, to "get data at this position", this ADT will need to access the underlying data structure used to implement `Book`. This may seem a bit odd, since we have gone to great lengths to hide that data structure from the rest of the world. But if you look back at our various declarations of the `Book` class, you can see that we declare the `AuthorPosition` *inside* the `Book` class. It's a part of `Book` and contributes to the support of the `Book` abstraction, giving it something of a privileged position.

Another consequence of this close relationship between the two classes is that the implementation of `AuthorPosition` will be different for each of the different implementations of `Book` that we have considered.

Simple Arrays

Simple Arrays

```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                          // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);
private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  static const int MAXAUTHORS = 12;
  Author authors[MAXAUTHORS];

};

#endif
```

If we implement `Book` using simple arrays, the implementation of `AuthorPosition` is trivial.

It takes advantage of the fact that the conventional interface for iterators in C++ is chosen to mimic the behavior of pointers to array elements.

. . . .

- Suppose we have an array `a`

  ```
  T a[N];
    ⋮
  for (int i = 0; i < N; i++)
     a[i] = foo(a[i]);
  ```

  - In C++, an array name without an index is actually a pointer to the first element:

    ```
    *a == a[0]
    ```

  - adding 1 to a pointer is equivalent to moving one element forward in an array:

    ```
    (a+1) == a[1], *(a+2) == a[2], etc.
    ```

So here is another way to iterate over an array:

```
T a[N];
 ⋮
for (T* p = a; p != a+N; p++)
   *p = foo(*p);
```

It does the exact same thing as the first form of iteration:

```
for (int i = 0; i < N; i++)
    a[i] = foo(a[i]);
```

C++ iterators are designed to mimic this behavior:

```
Container c;
 ⋮
for (iterator p = c.begin(); p != c.end(); p++)
    *p = foo(*p);
```

Iterators and Arrays

The only differences lie in how one gets access to the starting and ending positions.

	`T a[N];`	`Container c;`
get element at position	`*p p->`	`*it it->`
move forward 1 position	`++p`	`++it`

compare positions	`p1 == p2`
	`p1!= p2`
		`it1 == it2`
		`it1 != it2`

position of first element
position just after last element
	`a`
	`a+N`
		`c.begin()`
		`c.end()`

Consequently, we can actually implement the `AuthorPosition` ADT as a simple pointer to one of the array elements in `Book`'s array of authors.

Book (Array) Iterator

```
class Book {
public:
   typedef const Author* AuthorPosition;
     ⋮
};

Book::AuthorPosition Book::begin() const
{
 return authors;
}

Book::AuthorPosition Book::end() const
{
 return authors + numAuthors;
}
```

The pointer type already provides the *, ->, ++, ==, and != operations we need.

. . . .

So we just needed to implement the `begin()` and `end()` functions for `Book`.

Dynamically Allocated Arrays

Dynamically Allocated Arrays

```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;
  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }


  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  int MAXAUTHORS;
  Author* authors;

};

#endif
```

Everything we have said about iterators over simple arrays applies equally well to dynamic arrays. So the implementation of AuthorPostion is identical:

```
class Book {
public:
 typedef const Author* AuthorPosition;
    ⋮
```

. . . .

Linked Lists

Linked Lists

```
#ifndef BOOK_H
#include "author.h"
#include "authoriterator.h"
#include "publisher.h"


class Book {
public:
  typedef AuthorIterator AuthorPosition;

  Book (Author);                       // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors

  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  AuthorNode* first;
  AuthorNode* last;

  friend class AuthorIterator;
};
```

```
#endif
```

If the authors of a book are kept as a linked list, the iterator type requires a bit more work. The logical first thought for a way to designate a position within a linked list is to use a pointer to a linked list node. But, unlike the pointers to array elements, pointers to linked list nodes cannot serve as implementations of iterators. If `p` is a pointer to a linked list node, then `*p` returns the whole node, *not* the data stored inside that node, which is what we want from `*iterator`. Also, the operation `++p`, although it will compile, is technically illegal in C++ as increment of pointers is only legal within an array. What we *really* want `++iterator` to do is to follow the `next` link from one linked list node to another.

To get the desired behaviors for our iterator ADT, we will need to implement it as a class in its own right. The declaration shown here provides our desired interface.

```
class AuthorIterator {
public:
  AuthorIterator ();

  Author operator*() const;
  const Author* operator->() const;
  const AuthorIterator& operator++(); // prefix form ++i;
  AuthorIterator operator++(int); // postfix form i++;

  bool operator== (const AuthorIterator& ai) const;
  bool operator!= (const AuthorIterator& ai) const;

private:
  AuthorNode* pos;
  AuthorIterator (AuthorNode* p)
    : pos(p)
  {}

  friend class Book;
};
```

Inside the Book class, we write

```
class Book {
public:
   typedef AuthorIterator AuthorPosition;
     ⋮
 friend class AuthorIterator;
};
```

which gives AuthorPosition access to all private members of Book.

. . . .

Our AuthorIterator has one (hidden) data member, a pointer to a linked list node, which we use to indicate a position within the author list. But the operations we provide will interpret this pointer in a way that gives it iterator-appropriate behavior.

Implementing the Iterator Ops

For example, we implement the * operator like this:

```
Author AuthorIterator::operator*() const
{
  return pos->au;
}
```

returning the data field (only) from the linked list node.

. . . .

Implementing the Iterator Ops (cont.)

The other particularly interesting operator is ++:

```
// prefix form ++i;
const AuthorIterator& AuthorIterator::operator++()
{
  pos = pos->next;
  return *this;
}

// postfix form i++;
AuthorIterator AuthorIterator::operator++(int)
{
  AuthorIterator oldValue = *this;
  pos = pos->next; return oldValue;
}
```

The main thing that happens here is simple advancing the *pos* pointer to the next linked list node.

. . . .

Standard List

Standard Lists

```
#ifndef BOOK_H

#include <list>

#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef std::list<Author>::const_iterator AuthorPosition;
  typedef std::list<Author>::const_iterator const_AuthorPosition;

  Book (Author);                      // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors
  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  const_AuthorPosition begin() const;
  const_AuthorPosition end() const;

  AuthorPosition begin();
  AuthorPosition end();

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  std::list<Author> authors;
};
```

```
#endif
```

If we use a `std::list` to keep our authors, the implementation of our iterator becomes fairly simple again.

- That's because all the `std` containers provide their own iterators
  - (which, obviously, will follow the `std` interface), and so
  - we can simply implement our own iterator in terms of that one:

```
class Book {
public:
  typedef std::list<Author>::const_iterator AuthorPosition;
  typedef std::list<Author>::const_iterator const_AuthorPosition;
```

. . . .

The one complication is that we will actually need two iterator types, one for use with `const Book` objects and one for use with non-const `Books`. This is a common idiom in C++, which we will discuss later when we look at "const correctness".

Implementing the Iterator Ops

Implementing the `begin()` and `end()` functions is pretty straightforward.

```
Book::const_AuthorPosition Book::begin() const
{
  return authors.begin();
}

Book::const_AuthorPosition Book::end() const
{
  return authors.end();
}

Book::AuthorPosition Book::begin()
{
  return authors.begin();
}

Book::AuthorPosition Book::end()
{
  return authors.end();
}
```

. . . .

Chapter 5

Constructors and the Rule of the Big 3

Next we turn our attention to a set of issues that are often given short shrift in both introductory courses and textbooks, but that are extremely important in practical C++ programming.

As we begin to build up our own ADTS, implemented as C++ classes, we quickly come to the point where we need more than one of each kind of ADT object. Sometimes we will simply have multiple variables of our ADT types. Once we do, we will often want to copy or assign one variable to another, and *we need to understand what will happen when we do so*. Even more important, we need to be sure that what *does* happen is what we *want* to have happen, depending upon our intended behavior for our ADTs.

As we move past the simple case of multiple variables of the same ADT type, we may want to build *collections* of that ADT. The simplest case of this would be an array or linked list of our ADT type, though we will study other collections as the semester goes on. We will need to understand *what happens* when we initialize such a collection and when we copy values into and out of it, and we need to make sure that behavior is what we *want* for our ADTs.

In this lesson, we set the stage for this kind of understanding by looking at how we control initialization and copying of class values.

5.1 The Default Constructor

The Default Constructor

The *default constructor* is a constructor that takes no arguments. This is the constructor you are calling when you declare an object with no parameters. E.g.,

```
std::string s;
Book b;
```

. . . .

Declaration

A default constructor might be declared to take no parameters at all, like this:

```
class Book {
public:
   Book();
    ⋮
```

or with defaults on all of its parameters:

```
namespace std {
class string {
public:
    ⋮
   string(char* s = "");
    ⋮
```

Either way, we can *call* it with no parameters.

. . . .

Why Default?

Why is this called a "default" constructor? There's actually nothing particularly special about it. It's just an ordinary constructor, but it is *used* in a special way. Whenever we create an array of elements, the compiler implicitly calls this constructor to initialize each elements of the array.

For example, if we declared:

```
std::string words[5000];
```

then each of the 5000 elements of this array will be initialized using the default constructor for `string`

- In fact, if we don't *have* a default constructor for our class, then we *can't* create arrays containing that type of data. Attempting to do so will result in a compilation error.

. . . .

Compiler-Generated Default Constructors

Because arrays are so common, it is rare that we would actually want a class with no default constructor. The C++ compiler tries to be helpful:

If we create no constructors at all for a class, the compiler generates a default constructor for us.

- That automatically generated default constructor works by initializing every data member in our class with its own default constructor.
  - In many cases (e.g., strings), this does something entirely reasonable for our purposes.
  - Be careful, though: the "default constructors" for the primitive types such as `int`, `double` and pointers work by doing nothing at all. The value is left *uninitialized*.

. That leaves us with an uninitialized value containing whatever bits happened to have been left at that particular memory address by earlier calculations/programs.

5.1.1 Default constructor examples

Book default constructor

If we haven't created a default constructor for `Books`, we could add one easily enough:

```
class Book {
public:
  typedef AuthorIterator AuthorPosition;

  Book (Author);                           // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors
  Book();

  std::string getTitle() const        { return title; }
  void setTitle(std::string theTitle) {  title = theTitle; }
      ⋮
private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  AuthorNode* first;
  AuthorNode* last;

  friend class AuthorIterator;
};
```

To implement it, we need to come up with something reasonable for each data member:

```
Book::Book ()
   numAuthors(0), first(0), last(0)
{
}
```

(We'll let the strings `title` and `isbn` default to the `std::string` default of `""`, and we trust that the `Publisher` class provides a reasonable default constructor of its own.)

. . . .

Address default constructor

```
class Address {
public:
  Address (std::string theStreet, std::string theCity,
           std::string theState, std::string theZip);

  std::string getStreet() const;
  void putStreet (std::string theStreet);

  std::string getCity() const;
  void putCity (std::string theCity);

  std::string getState() const;
  void putState (std::string theState);

  std::string getZip() const;
  void putZip (std::string theZip);

private:
  std::string street;
  std::string city;
  std::string state;
  std::string zip;
};
```

This class has no explicit default constructor. It does have another constructor, however, so the compiler will not generate an automatic default constructor for us.

There's no good reason why we should not have a default constructor for this class. We might want arrays of addresses sometime.

```
class Address {
public:
  Address (std::string theStreet, std::string theCity,
           std::string theState, std::string theZip);

  Address () {}
    ⋮
```

As you can see, there's not much to it. All the data members are strings, and there's really nothing much better for us to do than just rely on the default constructors for strings.

. . . .

5.2 Copy Constructors

Copy Constructors

The *copy constructor* for a class `Foo` is the constructor of the form:

```
Foo (const Foo& oldCopy);
```

- Like the default constructor, there is nothing special about the copy constructor itself.
  - It's just an ordinary constructor.
- It's special because of the number of common situations in which it gets *used*.
  - Like the default constructor, the compiler often generates implicit calls to this constructor in places where we might not expect it.

. . . .

Where are Copy Constructors Used?

The copy constructor gets used in 5 situations:

1. When you declare a new object as a copy of an old one:

   ```
   Book book2 (book1);
   ```

   or

   ```
   Book book2 = book1;
   ```

2. When a function call passes a parameter “by copy” (i.e., the formal parameter does not have a &):

   ```
   void foo (Book b, int k);
     ⋮

   Book text361 (0201308787, budd,
        "Data Structures in C++ Using the Standard Template Library",
        1998, 1);
   foo (text361, 0);    // foo actually gets a copy of text361
   ```

3. When a function returns an object:

   ```
   Book foo (int k);
   {
     Book b;
     ⋮
     return b; // a copy of b is placed in the caller's memory area
   }
   ```

4. When data members are initialized in a constructor’s initialization list:

   ```
   Author::Author (std::string theName,
                   Address theAddress, long id)
   ```

```
  : name(theName),
    address(theAddress),
    identifier(id)
{
}
```

5. When an object is a data member of another class for which the compiler has generated its own copy constructor.

....

Compiler-Generated Copy Constructors

As you can see from that list, the copy constructor gets used a *lot*. It would be very awkward to work with a class that did not provide a copy constructor.

So, again, the compiler tries to be helpful.

If we do not create a copy constructor for a class, the compiler generates one for us.

- This automatically generated version works by copying each data member via *their* individual copy constructors.
- For data members that are primitives, such as `int` or pointers, the copying is done by copying all the bits of that primitive object.
  - For things like `int` or `double`, that's just fine.
  - We'll see shortly, however, that this may or may not be what we want for pointers.

....

5.2.1 Copy Constructors Examples

Address

Address Copy Constructor

In the case of our `Address` class, we have not provided a copy constructor, so the compiler would generate one for us.

The implicitly generated copy constructor would behave as if it had been written this way:

```
Address::Address (const Address& a)
  : street(a.street), city(a.city),
    state(a.state), zip(a.zip)
{
}
```

- This is, in fact, a perfectly good copy function for this class, so we might as well use the compiler-generated version.

. . . .

If our data members do not have explicit copy constructors (and *their* data members do not have explicit copy constructors, and . . .) then the compiler-provided copy constructor amounts to a bit-by-bit copy.

The compiler is all too happy to generate a copy constructor for us, but can we trust what it generates? To understand when we can and cannot trust it, we need to understand the different ways in which copying can occur.

Book - simple arrays

Book - simple arrays

```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const;
  void setTitle(std::string theTitle);

  int getNumberOfAuthors() const;

  std::string getISBN() const;
  void setISBN(std::string id);

  Publisher getPublisher() const;
  void setPublisher(const Publisher& publ);

  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  static const int MAXAUTHORS = 12;
  Author authors[MAXAUTHORS];

};

#endif
```

Consider the problem of copying a book, and for the moment we will work with our “simple” arrays version.

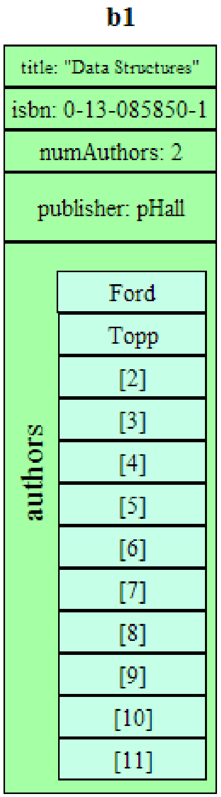
. . . .

Book - simple arrays (cont.)

If we start with a single book object, `b1`, as shown here, and then we execute

```
Book b2 = b1;
```

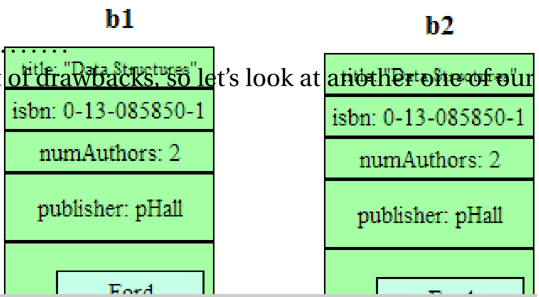
because we have provided no copy constructor, the compiler-generated version is used and each data member is copied.



. . . .

Compiler-Generated Copy

The new result would be something like this, which looks just fine.



. . . .

Of course, as we have noted before, this fixed-length array design has a lot of drawbacks, so let's look at another one of our implementations.

Book - dynamic arrays

Book - dynamic arrays

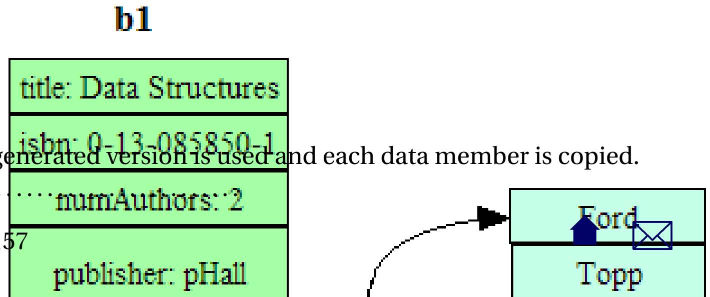
```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                     // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }
  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }


  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  int MAXAUTHORS;
  Author* authors;

};

#endif
```

Now let's consider the problem of copying a book implemented using dynamically allocated arrays.

If we start with a single book object, b1, as shown here, and then we execute

```
Book b2 = b1;
```

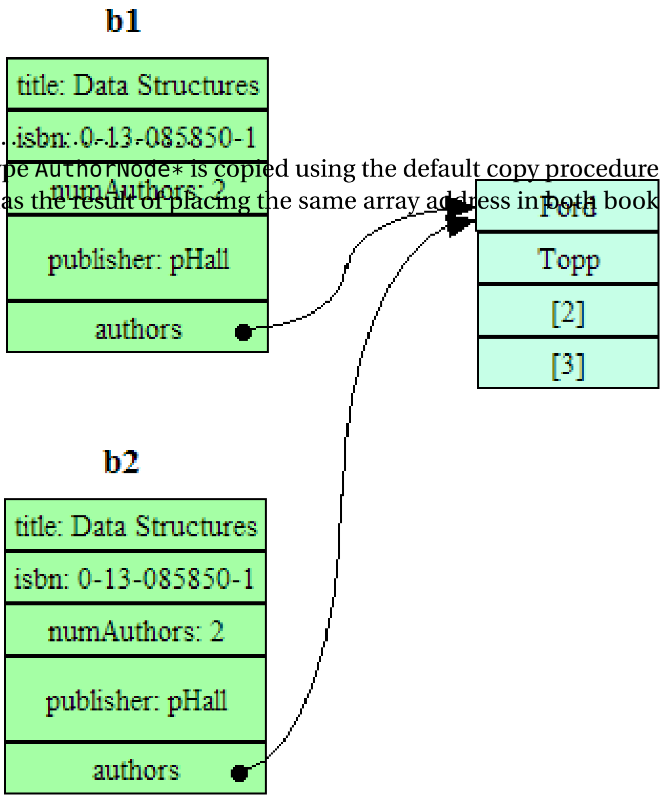
because we have provided no copy constructor, the compiler-generated version is used and each data member is copied.



Compiler-Generated Copy

The new result would be something like this.

- The *authors* pointer is copied bit-by-bit
- two book objects now share the same author array



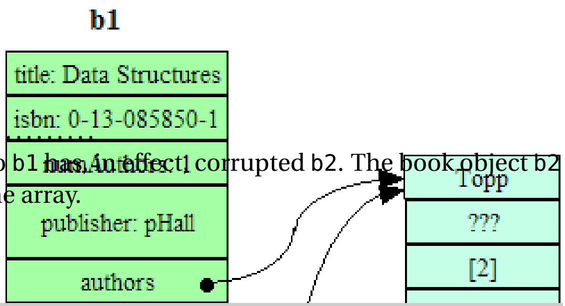
The key factor here is that the `authors` data member, of data type `AuthorNode*` is copied using the default copy procedure for pointers, which is to simply copy the bits of the pointer. That has the result of placing the same array address in both book objects. That's a really, really, bad idea!

Co-authors Should Not Share

To understand why this is so bad, suppose that we later did

```
b1.removeAuthor(b1.begin());
```

to remove the first author from book b1.



Afterwards, we would have something like this. Notice that the change to b1 has, in effect, corrupted b2. The book object b2 still believes it has two authors (numAuthors == 2) but there is only one in the array.

Unplanned Sharing Leads to Corruption

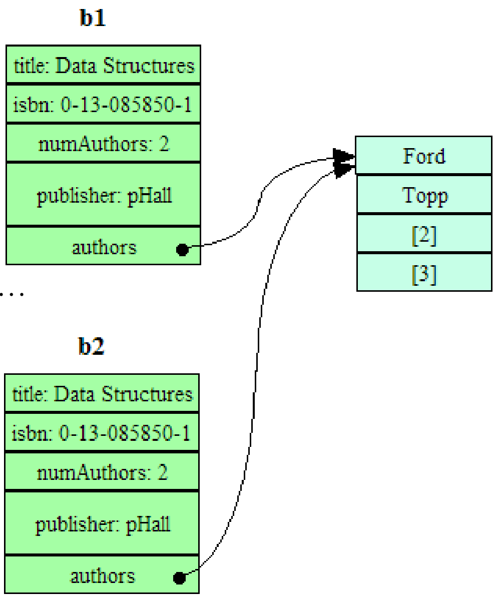
```
void Book::addAuthor (Book::AuthorPosition at, const Author& author)
{
  if (numAuthors >= MAXAUTHORS)
    {
      Author* newAuthors = new Author[2*MAXAUTHORS];
      for (int i = 0; i < MAXAUTHORS; ++i)
    newAuthors[i] = authors[i];
      MAXAUTHORS *= 2;
      delete [] authors;
      authors = newAuthors;
    }
  int i = numAuthors;
  int atk = at - authors;
  while (i > atk)
    {
      authors[i] = authors[i-1];
      i--;
    }
  authors[atk] = author;
  ++numAuthors;
}
```

That's not even the worst possible scenario. Recall that our code for adding authors works by

• Checking to see if there was room in the array.

• if not, 1. allocating a larger array, 2. copying the data, and 3. deleting the old array:

So if we start from this data state, and then add 3 authors to book 1 ("Doe", "Smith", and "Jones"), ...

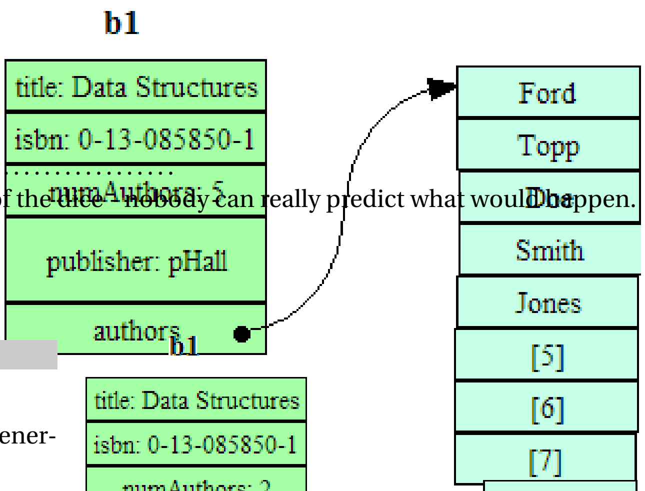


. . . .

Corrupted Data

… we would end up with the state shown here.

- `b1` deleted the old array,
- but `b2` still has the old array address in its data member.



So any attempt to access `b2`'s authors is now essentially a throw of the dice – nobody can really predict what would happen.

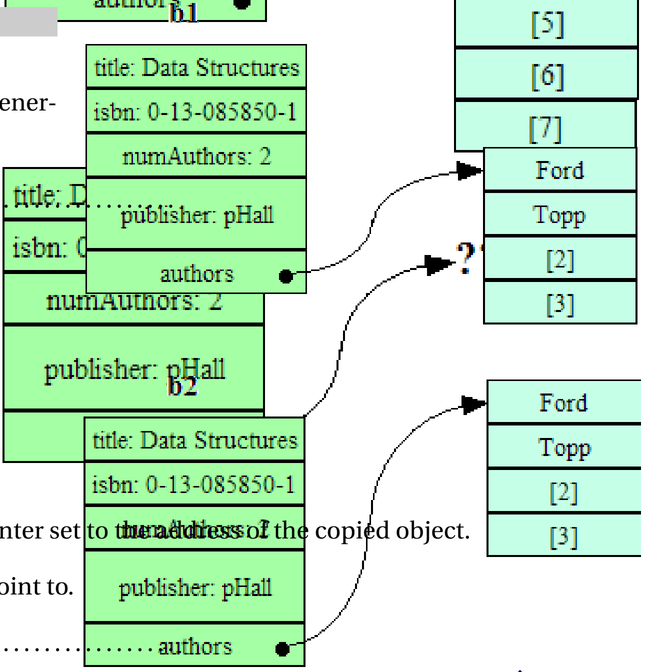
Sometime it's Better to Have 2 Copies

What we really wanted, after the copy:

```
Book b2 = b1;
```

is something more like this:

But to get that, we will not be able to rely on the automatically generated copy constructor for `Book`s.



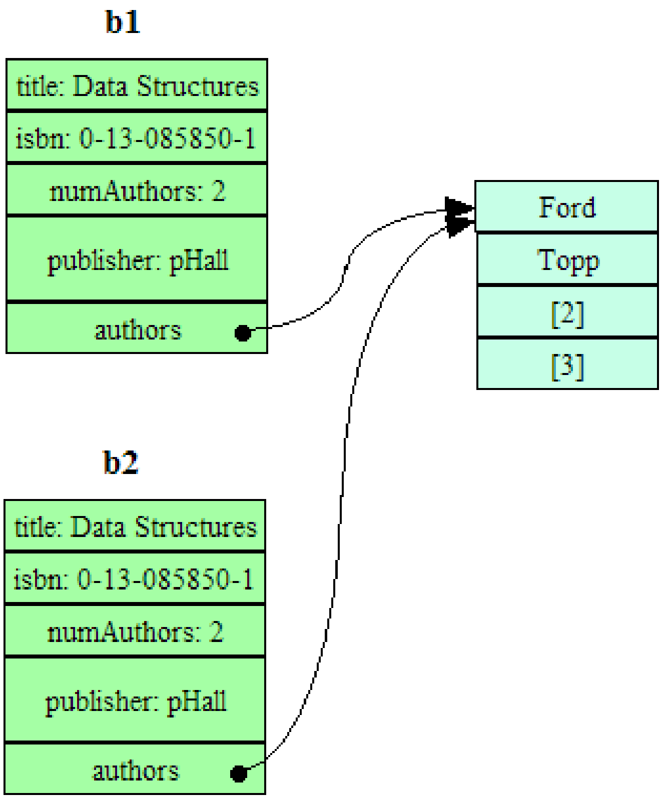
5.2.2 Shallow vs Deep Copy

Shallow vs Deep Copy

Copy operations are distinguished by how they treat pointers:

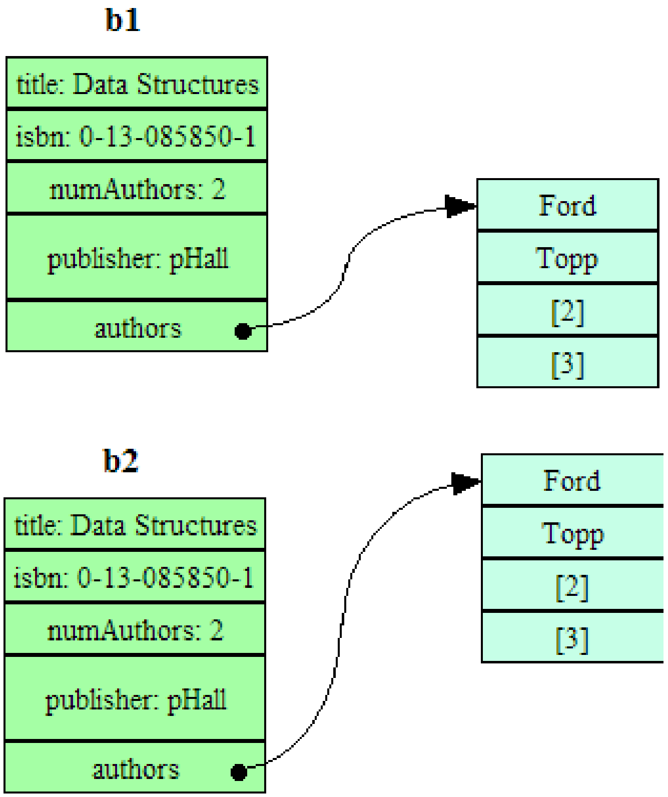
- In a *shallow copy*, all pointers are copied.
  - Leads to shared data on the heap.
- In a *deep copy*, objects pointed to are copied, then the new pointer set to the address of the copied object.
  - Copied objects keep exclusive access to the things they point to.

This was a Shallow Copy



. . . .

This was a Deep Copy



. . . .

Shallow versus Deep

- In this example, deep is preferred.
- That's not always the case.

  - When we design an ADT we have to *think* about what is right for the abstraction we want to provide

- "Shallow" and "deep" are two extremes of a range of possible copies

. . . .

For any ADT, we must decide whether the things it points to are things we want to *share* with other objects or whether we want to *own* them exclusively. That's a matter of just how we want our ADTs to behave, which depends in turn on what we expect to do with them.

In this context, it should be noted that "shallow" and "deep" are actually two extremes of a range of possible copy depths - sometimes our ADTs call for a behavior that has us treat some pointers shallowly and others deeply.

5.2.3 Deep Copy Examples

Book - dynamic arrays

If we want to implement a proper deep copy for our books, we start by adding the constructor declaration:

```
class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors

  Book (const Book& b);

  std::string getTitle() const        { return title; }
  void setTitle(std::string theTitle) {  title = theTitle; }
     ⋮
private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  int MAXAUTHORS;
  Author* authors;

};
```

Then we supply a function body for this constructor.

```
Book::Book (const Book& b)
  : title(b.title), isbn(b.isbn), publisher(b.publisher),
    numAuthors(b.numAuthors), MAXAUTHORS(b.MAXAUTHORS)
{
  authors = new Author[numAuthors+1];
  for (int i = 0; i < numAuthors; ++i)
    authors[i] = b.authors[i];
}
```

. . . .

Most of the data members can be copied easily. But the `authors` pointer is copied by allocating a new array big enough to hold the existing data; then all the existing data has to be copied into the new array.

Book - linked list

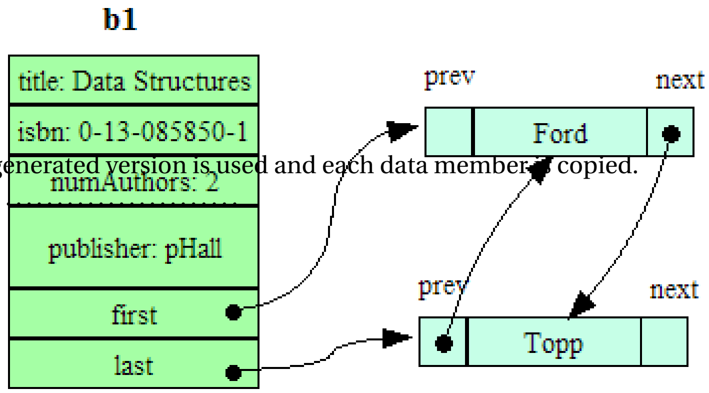
```
#ifndef BOOK_H
#include "author.h"
#include "authoriterator.h"
#include "publisher.h"


class Book {
public:
  typedef AuthorIterator AuthorPosition;

  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;
  AuthorNode* first;
  AuthorNode* last;

  friend class AuthorIterator;
};

#endif
```

Now let's consider the problem of copying a book implemented using linked lists.

If we start with a single book object, b1, as shown here, and then we execute

```
Book b2 = b1;
```

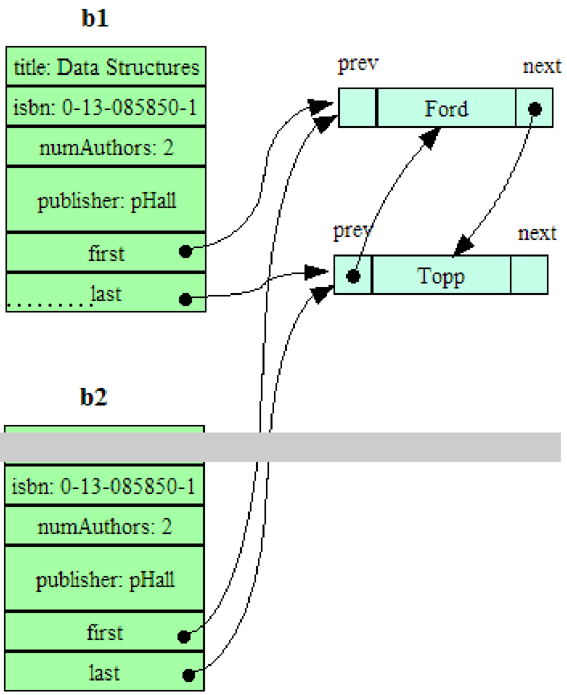
because we have provided no copy constructor, the compiler-generated version is used and each data member is copied.



Shallow Copy of Linked List

The new result would be something like this.

- Again, the pointer data members (`first` and `last`) are copied using the default copy procedure for pointers, which is to simply copy the bits of the pointer.
- That has the result of placing the same node addresses in both book objects. In effect, there is one collection of nodes being shared between both books.
- Once again, that's a really, really, bad idea!



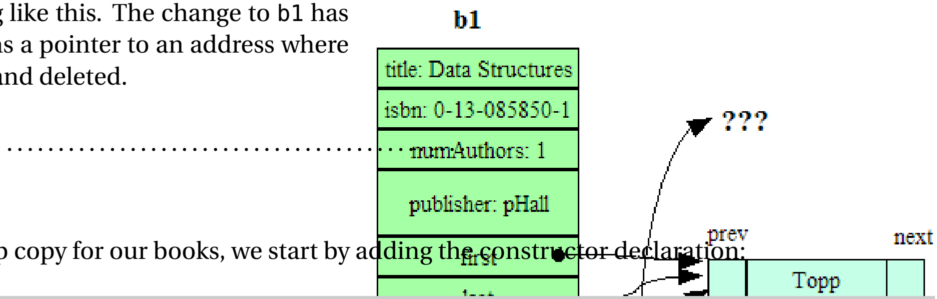
Corrupted List

For example, suppose that we later did

```
b1.removeAuthor(b1.begin());
```

to remove the first author from book `b1`.

Afterwards, we would have something like this. The change to b1 has corrupted b2. The book object b2 contains a pointer to an address where the node has been unlinked from the list and deleted.



List-Based Copy Constructor

If we want to implement a proper deep copy for our books, we start by adding the constructor declaration:

```
class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors

  Book (const Book& b);

  std::string getTitle() const        { return title; }
  void setTitle(std::string theTitle) {  title = theTitle; }
     ⋮
private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  AuthorNode* first;
  AuthorNode* last;
  friend class AuthorIterator;
};
```

Then we supply a function body for this constructor.

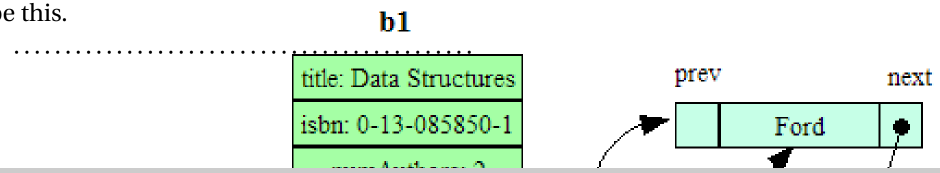
```
Book::Book (const Book& b)
  : title(b.title), isbn(b.isbn),
    publisher(b.publisher),
    numAuthors(0), first(0), last(0)
{
  for (AuthorPosition p = b.begin(); p != b.end();
       ++p)
    addAuthor(end(), *p);
}
```

. . . .

Most of the data members can be copied easily. But the `first` and `last` pointers are copied by setting the list up first as an empty list, then copying each author from the existing book into the new one.

Linked-List Deep Copy

The end result after the copy should be this.



Book - std::list

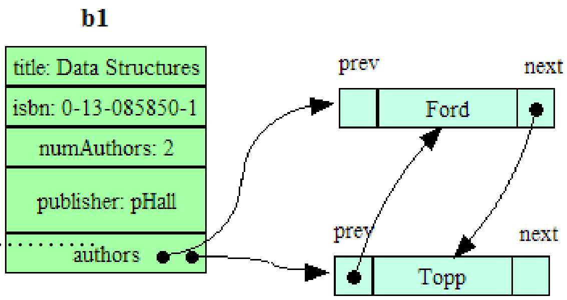
```
#ifndef BOOK_H

#include <list>

#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef std::list<Author>::iterator AuthorPosition;
  typedef std::list<Author>::const_iterator const_AuthorPosition;

  Book (Author);                                 // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }
  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  const_AuthorPosition begin() const;
  const_AuthorPosition end() const;

  AuthorPosition begin();
  AuthorPosition end();

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  std::list<Author> authors;
};

#endif
```

Now let's consider the problem of copying a book implemented using `std::list`.

If we start with a single book object, `b1`, as shown here, and then we execute

```
Book b2 = b1;
```

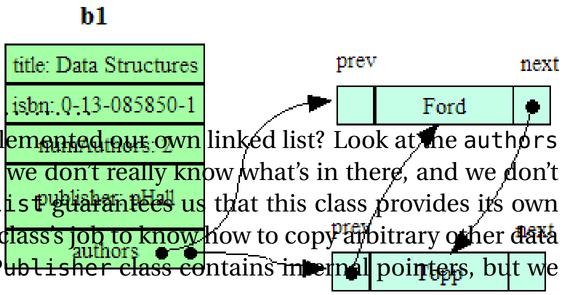
because we have provided no copy constructor, the compiler-generated version is used and each data member is copied.



Copy of std::list

The new result would be something like this. This looks just fine.

We don't need to override the compiler-generated copy constructor in this case.



But what's so different between this case and the last one where we implemented our own linked list? Look at the `authors` data member. This is not a pointer. It's an object of type `std::list`. Now, we don't really know what's in there, and we don't need to (or particularly want to) know. But the documentation for `std::list` guarantees us that this class provides its own deep copy for the copy constructor, so we trust it. It's certainly not the `Book` class's job to know how to copy arbitrary other data structures. (For that matter, we have no idea at the moment whether the `Publisher` class contains internal pointers, but we trust that class to also provide a proper copy constructor.)

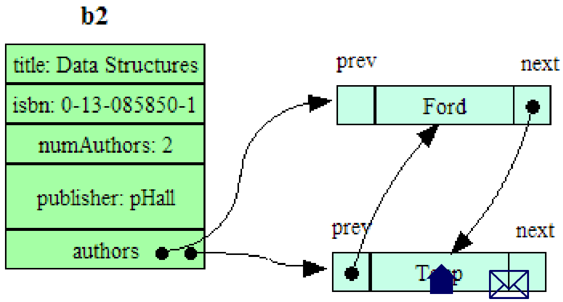
5.2.4 Trusting the Compiler-Generated Copy Constructor

Trusting the Compiler-Generated Copy Constructor

By now, you may have perceived a pattern.

Shallow copy is wrong when...

- Your ADT has pointers among its data members, and
- You don't want to share the objects being pointed to.



And it follows that:
Compiler-generated copy constructors are wrong when...

- Your ADT has pointers among its data members, and
- You don't want to share the objects being pointed to.

. . . .

5.3 Assignment

Assignment

When we write `book1 = book2`, that's shorthand for `book1.operator=(book2)`.
We tend to do a lot of assignment in typical programming, so, once more, the compiler tries to be helpful:

If you don't provide your own assignment operator for a class, the compiler generates one automatically.

- The automatically generated assignment operator works by assigning each data member in turn.
- If none of the members have programmer-supplied assignment ops, then this is a *shallow copy*

. . . .

A Compiler-Generated Assignment Op

```
class Address {
public:
  Address (std::string theStreet, std::string theCity,
           std::string theState, std::string theZip);

  std::string getStreet() const;
  void putStreet (std::string theStreet);

  std::string getCity() const;
  void putCity (std::string theCity);

  std::string getState() const;
  void putState (std::string theState);

  std::string getZip() const;
  void putZip (std::string theZip);

private:
  std::string street;
  std::string city;
  std::string state;
  std::string zip;
};
```

For example, we have not provided an assignment operator for `Address` class. Therefore the compiler will attempt to generate one, just as if we had written

```
class Address {
public:
  Address (std::string theStreet, std::string theCity,
           std::string theState, std::string theZip);


  Address& operator= (const Address&);
     ⋮
```

. . . .

A Compiler-Generated Assignment Op (cont.)

The automatically generated body for this assignment operator will be the equivalent of

```
Address& Address::operator= (const Address& a)
{
  street = a.street;
  city = a.city;
  state = a.state;
  zip = a.zip;
  return *this;
}
```

And that automatically generated assignment is just fine for `Address`.

. . . .

Return values in Asst Ops

```
Address& Address::operator= (const Address& a)
{
  street = a.street;
  city = a.city;
  state = a.state;
  zip = a.zip;
  return *this;
}
```

The return value returns the value just assigned, allowing programmers to chain assignments together:

```
addr3 = addr2 = addt1;
```

This can simplify code where a computed value needs to be tested and then maybe used again if it passes the test, e.g.,

```
while ((x = foo(y)) > 0) {
  do_something_useful_with(x);
}
```

. . . .

The compiler is all too happy to generate an assignment operator for us, but can we trust what it generates? To understand when we can and cannot trust it, we need to understand the different ways in which copying can occur.

5.3.1 Assignment Examples

Book - Simple Arrays

Book - Simple Arrays

```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                            // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const;
  void setTitle(std::string theTitle);

  int getNumberOfAuthors() const;

  std::string getISBN() const;
  void setISBN(std::string id);

  Publisher getPublisher() const;
  void setPublisher(const Publisher& publ);

  AuthorPosition begin() const;
  AuthorPosition end() const;
  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  static const int MAXAUTHORS = 12;
  Author authors[MAXAUTHORS];

};

#endif
```

Consider the problem of copying a book, and for the moment we will work with our “simple” arrays version.

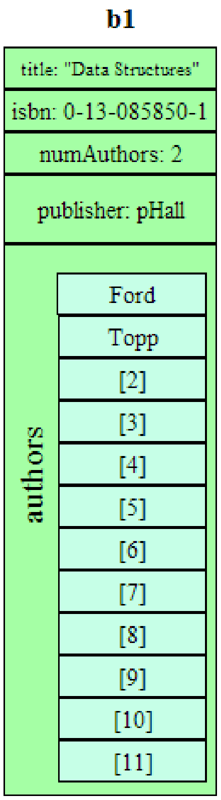
. . . .

Book - Simple Arrays (cont.)

If we start with a single book object, `b1`, as shown here, and then we execute

```
b2 = b1;
```

because we have provided no assignment operator, the compiler-generated version is used and each data member is copied.

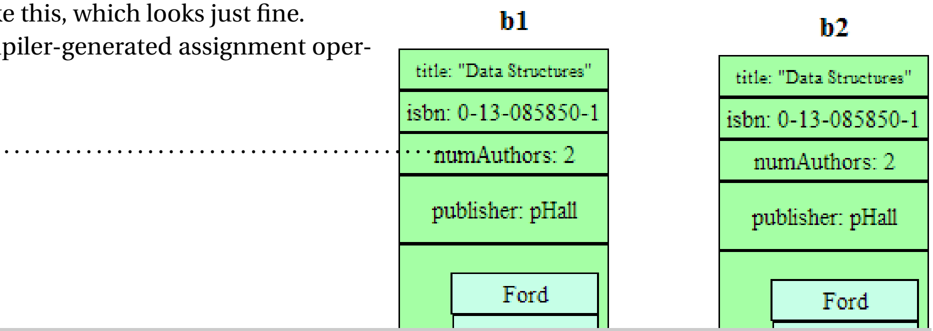


. . . .

Compiler-Generated Assignment

The new result would be something like this, which looks just fine.

So in this case, we can rely on the compiler-generated assignment operator.



Book - dynamic arrays

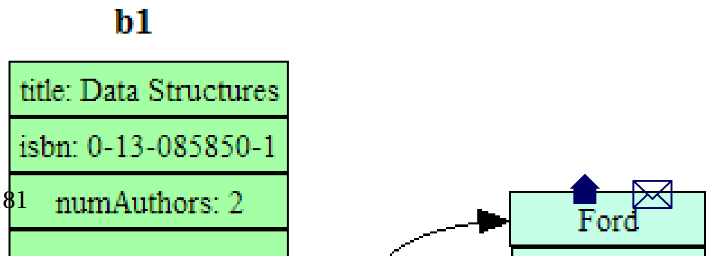
Book - dynamic arrays

```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                       // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }
  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }


  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  int MAXAUTHORS;
  Author* authors;

};

#endif
```

Now let's consider the problem of copying a book implemented using dynamically allocated arrays.

If we start with a single book object, b1, as shown here, and then we execute

```
b2 = b1;
```



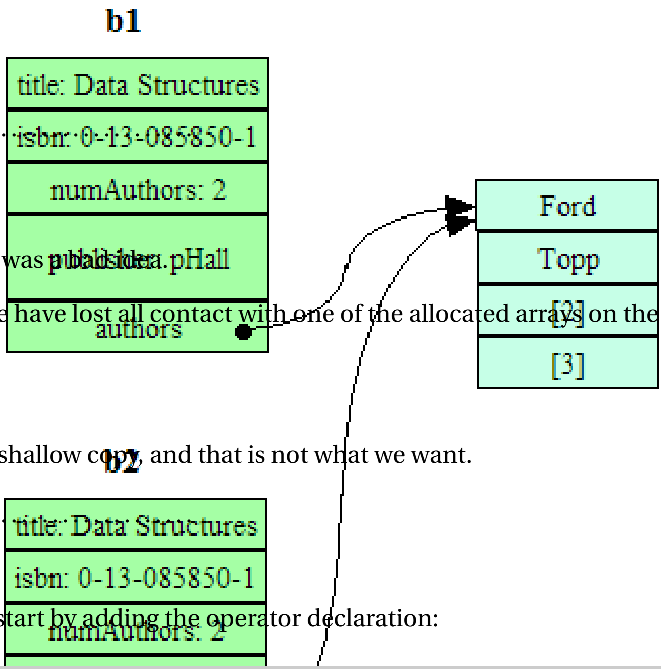
because we have provided no assignment op, the compiler-generated version is used and each data member is copied.

. . . .

Compiler-Generated Asst

The new result would be something like this.

- The *authors* pointer is copied bit-by-bit
- two book objects now share the same author array



. . . .

This Seems Familiar...

- We saw earlier that having two books sharing the same array was a bad idea.
- In this case, we now have the additional disadvantage that we have lost all contact with one of the allocated arrays on the heap,
  - a "memory leak" that will never be recovered.
- The compiler-generated assignment operator implements a shallow copy, and that is not what we want.

. . . .

Implementing Deep-Copy Assignment

If we want to implement a proper deep copy for our books, we start by adding the operator declaration:

```
class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                     // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors

  Book (const Book& b);
  const Book& operator= (const Book& b);

  std::string getTitle() const        { return title; }
  void setTitle(std::string theTitle) {  title = theTitle; }
     ⋮
private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  int MAXAUTHORS;
  Author* authors;

};
```

Then we supply a function body for this operator.

```
const Book& Book::operator= (const Book& b)
{
  title = b.title;
  isbn = b.isbn;
  publisher = b.publisher;
  numAuthors = b.numAuthors;
  if (b.numAuthors > MAXAUTHORS)
    {
      MAXAUTHORS = b.MAXAUTHORS;
      delete [] authors;
      authors = new Author[MAXAUTHORS];
    }
  for (int i = 0; i < numAuthors; ++i)
    authors[i] = b.authors[i];
  return *this;
}
```

Most of the data members can be copied easily. But the `authors` pointer is copied by allocating a new array big enough to hold the existing data; then all the existing data has to be copied into the new array.

Book - linked list

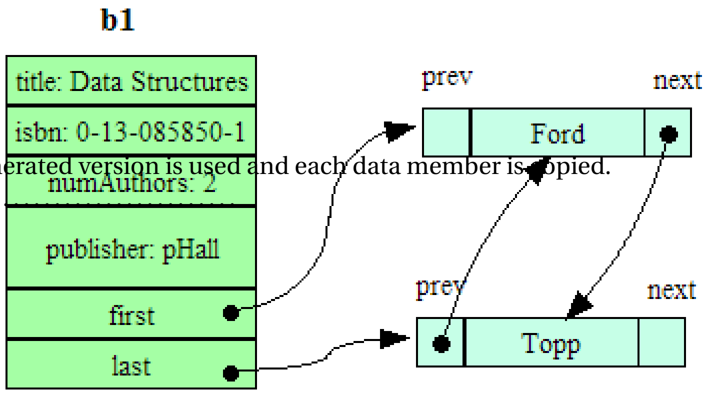
Book - linked list

Now let's consider the problem of copying a book implemented using linked lists.

If we start with a single book object, `b1`, as shown here, and then we execute

```
b2 = b1;
```

because we have provided no assignment op, the compiler-generated version is used and each data member is copied.



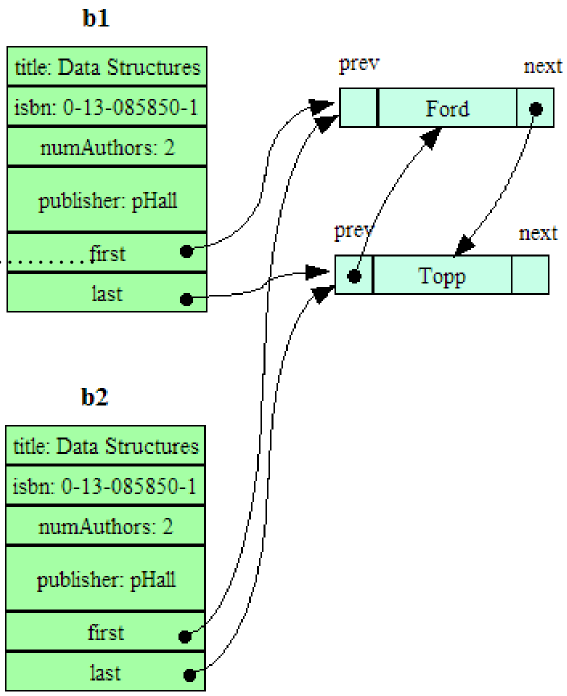
Shallow Copy of Linked List

The new result would be something like this.

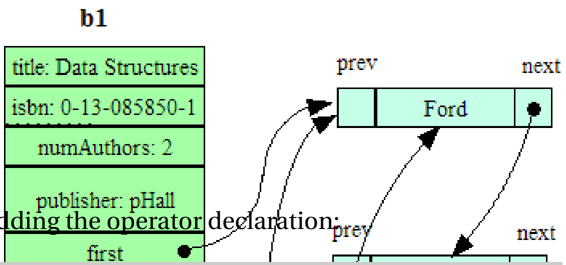
• Again, the pointer data members (`first` and `last`) are cassigned using the default assignment procedure for pointers, which is to simply copy the bits of the pointer.

• That has the result of placing the same node addresses in both book objects. In effect, there is one collection of nodes being shared between both books.

Shallow Copy of Linked List



- Once again, that's a really, really, bad idea!
- And, once again, the original nodes in b2 are now unreachable memory leaks.



Assignment via Linked Lists

If we want to implement a proper deep copy for our books, we start by adding the operator declaration:

```
class Book {
public:
  typedef const Author∗ AuthorPosition;

  Book (Author);                          // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors

  Book (const Book& b);
  const Book& operator= (const Book& b);

  std::string getTitle() const        { return title; }
  void setTitle(std::string theTitle) {  title = theTitle; }
     ⋮
private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  AuthorNode∗ first;
  AuthorNode∗ last;
  friend class AuthorIterator;
};
```

Then we supply a function body for this operator.

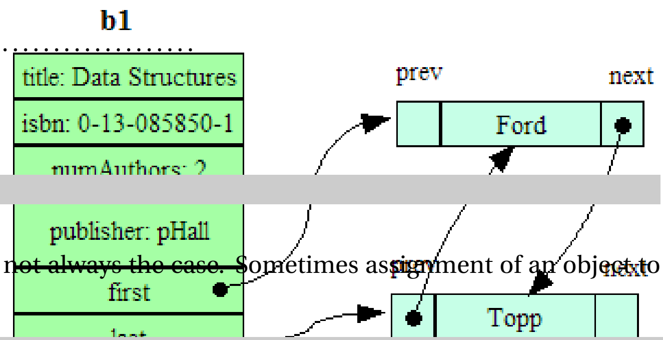
```
const Book& Book::operator= (const Book& b)
{
  title = b.title;
  isbn = b.isbn;
  publisher = b.publisher;
  numAuthors = 0;
  for (AuthorPosition p = begin(); p != end();)
    {
      AuthorPosition nxt = p;
      ++nxt;
      delete p;
      p = nxt;
    }
  first =  last = 0;
  for (AuthorPosition p = b.begin(); p != b.end(); ++p)
    addAuthor(end(), *p);
  return *this;
}
```

. . . .

Most of the data members can be copied easily. But the `first` and `last` pointers are copied by first emptying out any elements already in the list, then copying each author from the source book into the new one.

Assignment Result

The end result after the assignment should be this:



Self-Assignment

If we assign something to itself:

```
x = x;
```

we normally expect that nothing really happens.

But when we are writing our own assignment operators, that's not always the case. Sometimes assignment of an object to itself is a nasty special case that breaks thing badly.

```
const Book& Book::operator= (const Book& b)
{
  title = b.title;
  isbn = b.isbn;
  publisher = b.publisher;
  numAuthors = 0;
  for (AuthorPosition p = begin(); p != end();)
    {
      AuthorPosition nxt = p;
      ++nxt;
      delete p;
      p = nxt;
    }
  first =  last = 0;
  for (AuthorPosition p = b.begin(); p != b.end(); ++p)
    addAuthor(end(), *p);
  return *this;
}
```

What happens if we do `b1 = b1;`?

Self-Assignment Can Corrupt

- The first loop removes all nodes from the destination list.
- The second loop copies all nodes in the source list.
- But if the source and destination are the same, then by the time we reach the second loop, there won't be any nodes left to copy.

So, instead of `b1 = b1;` leaving `b1` unchanged, it would actually destroy `b1`.

. . . .

Checking for Self-Assignment

```
const Book& Book::operator= (const Book& b)
{
  if (this != &b)
    {
      title = b.title;
      isbn = b.isbn;
      publisher = b.publisher;
      numAuthors = 0;
      for (AuthorPosition p = begin(); p != end();)
        {
          AuthorPosition nxt = p;
          ++nxt;
          delete p;
          p = nxt;
        }
      first =  last = 0;
      for (AuthorPosition p = b.begin(); p != b.end(); ++p)
         addAuthor(end(), *p);
      }
  return *this;
}
```

This is safer.

- We check to see if the object we are assigning *to* (`this`) is at the same address as the one we are assigning *from*
  - the & in the expression `&b` is the C++ *address-of* operator).
- If the two are the same, we leave them alone.
- Only if the two addresses are different do we carry on with the assignment.

. . . .

You might think that self-assignment is so rare that we wouldn't need to worry about it. But, in practice, you might have lots of ways to reach the same object. For example, we might have passed the same object as two different parameters of a function call `foo(b1,b1)`. If the function body of `foo` were to assign one parameter to another, we would then have a self-assignment that would likely not have been anticipated by the author of `foo` and that would be very hard to detect in the code that called `foo`. As another example, algorithms for sorting arrays often contain statements like

```
array[i] = array[j];
```

with a very real possibility that, on occasion, `i` and `j` might be equal.

So self-assignment does occur in practice, and it's a good idea to check for this whenever you write your own assignment operators.

Book - std::list

Book - std::list

```
#ifndef BOOK_H

#include <list>
```

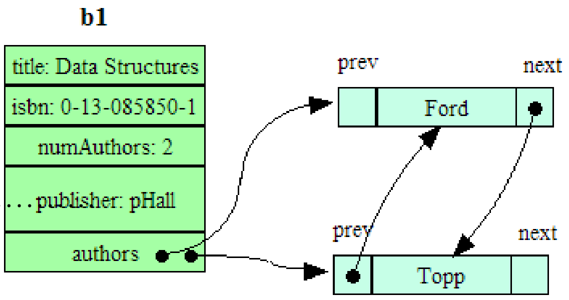
```
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef std::list<Author>::iterator AuthorPosition;
  typedef std::list<Author>::const_iterator const_AuthorPosition;

  Book (Author);                      // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  const_AuthorPosition begin() const;
  const_AuthorPosition end() const;

  AuthorPosition begin();
  AuthorPosition end();
  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  std::list<Author> authors;
};

#endif
```

Now let's consider the problem of assigning books implemented using `std::list`.

If we start with a single book object, `b1`, as shown here, and then we execute

```
b2 = b1;
```

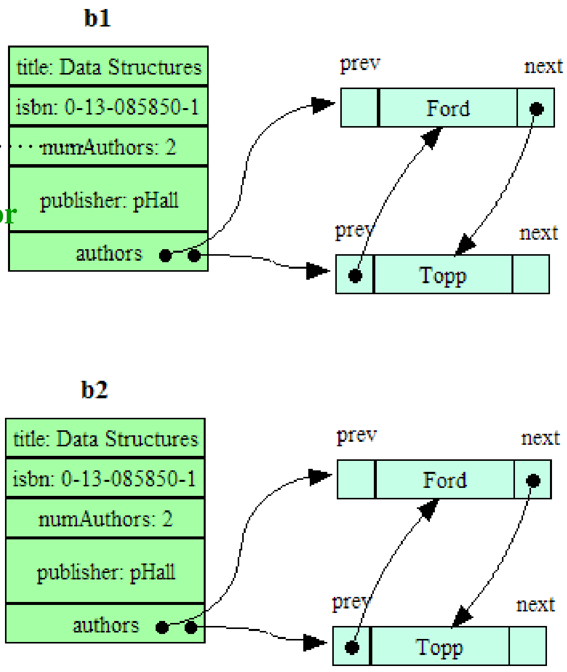
because we have provided no asst op, the compiler-generated version is used and each data member is copied.



Assigned std::list

The new result would be something like this. This looks just fine.

We don't need to override the compiler-generated assignment operator in this case.



5.3.2 Trusting the Compiler-Generated Assignment Operator

Trusting the Compiler-Generated Assignment Operator

By now, you may have perceived a pattern.

Shallow copy is wrong when...

- Your ADT has pointers among its data members, and
- You don't want to share the objects being pointed to.

And it follows that:

Compiler-generated assignment operators are wrong when...

- Your ADT has pointers among its data members, and
- You don't want to share the objects being pointed to.

5.4 Destructors

Destructors

Destructors are used to clean up objects that are no longer in use.

If you don't provide a destructor for a class, the compiler generates one for you automatically.

- The automatically generated destructor simply invokes the destructors for any data member objects.

- If none of the members have programmer-supplied destructors, does nothing.

. . . .

A Compiler-Generated Destructor

```
class Address {
public:
  Address (std::string theStreet, std::string theCity,
           std::string theState, std::string theZip);

  std::string getStreet() const;
  void putStreet (std::string theStreet);

  std::string getCity() const;
  void putCity (std::string theCity);

  std::string getState() const;
  void putState (std::string theState);

  std::string getZip() const;
  void putZip (std::string theZip);

private:
  std::string street;
  std::string city;
  std::string state;
  std::string zip;
};
```

```
class Author
{
public:
  Author (std::string theName, Address theAddress, long id);

  std::string getName() const        {return name;}
  void putName (std::string theName) {name = theName;}

  const Address& getAddress() const   {return address;}
  void putAddress (const Address& addr) {address = addr;}

  long getIdentifier() const     {return identifier;}

private:
  std::string name;
  Address address;
  const long identifier;
};
```

We have not declared or implemented a destructor for any of our classes. For `Address` and `Author`, that's OK.

- Note that the *string*s probably do contain pointers, but we trust the `std::string` to handle its own cleanup.

. . . .

5.4.1 Destructor Examples

Book - simple arrays

Book - simple arrays

```
#ifndef BOOK_H
#include "author.h"
```

```
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                          // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const;
  void setTitle(std::string theTitle);

  int getNumberOfAuthors() const;

  std::string getISBN() const;
  void setISBN(std::string id);

  Publisher getPublisher() const;
  void setPublisher(const Publisher& publ);

  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  static const int MAXAUTHORS = 12;
  Author authors[MAXAUTHORS];

};

#endif
```

This version of the book has all of its data in a single block of memory. Assuming that each data member knows how to clean up its own internal storage, there's really nothing we would have to do when this book gets destroyed.

We can rely on the compiler-provided destructor.

....

Book - dynamic arrays

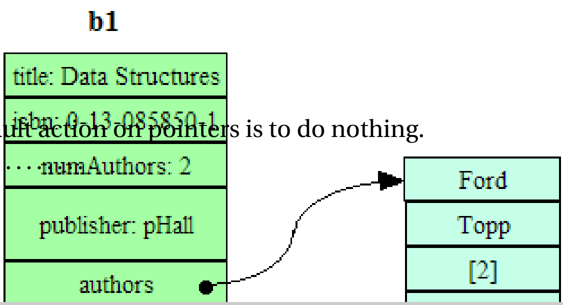
Book - dynamic arrays

```
#ifndef BOOK_H
#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                       // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors

  Book (const Book& b);
  ~Book();
  const Book& operator= (const Book& b);


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  AuthorPosition begin() const;
  AuthorPosition end() const;

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  int MAXAUTHORS;
  Author* authors;

};

#endif
```

In this version of the Book class, a portion of the data is kept on the heap, in a number of linked list nodes allocated on the heap.

• If this object were destroyed, we would need to be sure that the storage allocated for the array is recovered.

• That won't happen in the compiler-generated destructor, because the default action on pointers is to do nothing.



Implementing the Dyn. ArrayDestructor

We start by adding the destructor declaration:

```
class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                         // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors

  Book (const Book& b);
  const Book& operator= (const Book& b);
  ~Book();

  std::string getTitle() const        { return title; }
  void setTitle(std::string theTitle) {  title = theTitle; }
     ⋮
private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  int MAXAUTHORS;
  Author* authors;

};
```

```
Book::~Book()
{
  delete [] authors;
}
```

Then we supply a function body for this destructor. Not much needs to be done - just delete the pointer to the array of authors.

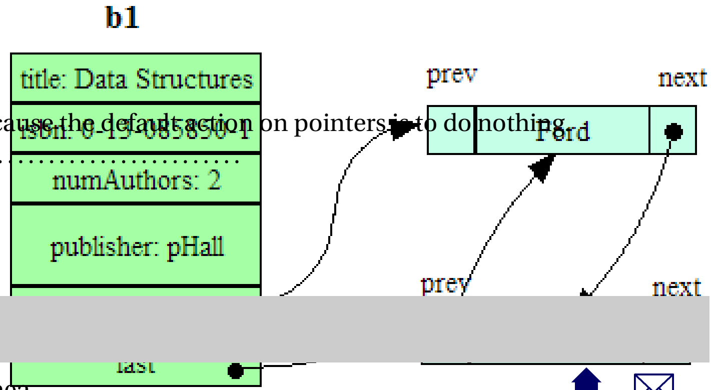
. . . .

Book - linked list

Book - linked list

```
#ifndef BOOK_H
#include "author.h"
#include "authoriterator.h"
#include "publisher.h"


class Book {
public:
  typedef AuthorIterator AuthorPosition;

  Book (Author);                        // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors

  Book (const Book& b);
  ~Book();
  const Book& operator= (const Book& b);

  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }

  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  AuthorPosition begin() const;
  AuthorPosition end() const;
  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  AuthorNode* first;
  AuthorNode* last;

  friend class AuthorIterator;
};

#endif
```

In this version of the Book class, a portion of the data is kept in an array allocated on the heap.



• If this object were destroyed, we would need to be sure that the storage allocated for the nodes is recovered.

• That won't happen in the compiler-generated destructor, because the default action on pointers is to do nothing.

Implementing a Linked List Destructor

We start by adding the destructor declaration:

```
class Book {
public:
  typedef const Author* AuthorPosition;

  Book (Author);                           // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors

  Book (const Book& b);
  const Book& operator= (const Book& b);
  ~Book();

  std::string getTitle() const        { return title; }
  void setTitle(std::string theTitle) {  title = theTitle; }
     ⋮
private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  AuthorNode* first;
  AuthorNode* last;

  friend class AuthorIterator;
};
```

Then we supply a function body for this destructor.

```
Book::~Book()
{
  AuthorPosition nxt;
  for (AuthorPosition current = begin(); current != end(); current = nxt)
    {
      nxt = current;
      ++nxt;
      delete current.pos;
    }
}
```

This one is more elaborate. We need to walk the entire list, deleting each node as we come to it.

. . . .

Book - std::list

Book - std::list

```
#ifndef BOOK_H

#include <list>

#include "author.h"
#include "publisher.h"


class Book {
public:
  typedef std::list<Author>::iterator AuthorPosition;
  typedef std::list<Author>::const_iterator const_AuthorPosition;

  Book (Author);                      // for books with single authors
  Book (const Author[], int nAuthors); // for books with multiple authors


  std::string getTitle() const        { return title; }

  void setTitle(std::string theTitle) {  title = theTitle; }
  int getNumberOfAuthors() const { return numAuthors; }

  std::string getISBN() const  { return isbn; }
  void setISBN(std::string id) { isbn = id; }

  Publisher getPublisher() const { return publisher; }
  void setPublisher(const Publisher& publ) { publisher = publ; }

  const_AuthorPosition begin() const;
  const_AuthorPosition end() const;

  AuthorPosition begin();
  AuthorPosition end();

  void addAuthor (AuthorPosition at, const Author& author);
  void removeAuthor (AuthorPosition at);

private:

  std::string title;
  int numAuthors;
  std::string isbn;
  Publisher publisher;

  std::list<Author> authors;
};

#endif
```

- This version of the book has no pointers among its data members.

- It's entirely likely that the `std::list` has pointers inside it, but we trust its destructor to clean up its own internal data structures.

  With that in mind, there's really nothing we would have to do when this book gets destroyed. We can rely on the compiler-provided destructor.

. . . .

5.4.2 Trusting the Compiler-Generated Destructor

Trusting the Compiler-Generated Destructor

By now, you may have perceived a pattern.
Compiler-generated destructors are wrong when...

- Your ADT has pointers among its data members, and
- You don't want to share the objects being pointed to.

. . . .

5.5 The Rule of the Big 3

The Big 3

The "*Big 3*" are the

- copy constructor
- assignment operator, and
- destructor

By now, you may have noticed a pattern with these.

. . . .

The Rule of the Big 3

The *rule of the big 3* states that,

> *if you provide your own version of any one of the big 3, you should provide your own version of all 3.*

Why? Because we don't trust the compiler-generated...

- copy constructor if our data members include pointers to data we don't share
- assignment operator if our data members include pointers to data we don't share
- destructor if our data members include pointers to data we don't share

So if we don't trust one, we don't trust any of them.

....

Chapter 6

A C++ Class Designer's Checklist

6.1 The Checklist

The Checklist

1. Is the interface complete?
2. Are there redundant functions in the interface that could be removed? Are there functions that could be generalized?
3. Have you used names that are meaningful in the application area?
4. Are pre-conditions and assumptions well documented? Have you used `assert` to "guard" the inputs?
5. Are the data members private?
6. Does every constructor initialize every data member?
7. Have you appropriately treated the default constructor?
8. Have you appropriately treated the "big 3" (copy constructor, assignment operator, and destructor)?

9. Does your assignment operator handle self-assignment?

10. Does your class provide == and < operators?

11. Does your class provide an output routine?

12. Is your class const-correct?

. . . .

Purpose

This is a *checklist*

- Use it whenever you have the responsibility of designing an ADT interface
- These are not absolute rules, but are things that you need to think about
  - Violate them if necessary, but only after careful consideration

. . . .

6.2 Discussion and Explanation

6.2.1 Is the interface complete?

Is the interface complete?

An ADT interface is *complete* if it contains all the operations required to implement the application at hand (and/or reasonably probable applications in the near future).

The best way to determine this is to look to the requirements of the application.

. . . .

Is Day complete?

```
class Day
{
   /**
      Represents a day with a given year, month, and day
      of the Gregorian calendar. The Gregorian calendar
      replaced the Julian calendar beginning on
      October 15, 1582
   */

public:

  Day(int aYear, int aMonth, int aDate);

   /**
      Returns the year of this day
      @return the year
   */
  int getYear() const;


  /**
     Returns the month of this day
     @return the month
  */
  int getMonth() const;

  /**
     Returns the day of the month of this day
     @return the day of the month
  */
  int getDate() const;


  /**
     Returns a day that is a certain number of days away from
     this day
     @param n the number of days, can be negative
     @return a day that is n days away from this one
  */
  Day addDays(int n) const;


  /**
     Returns the number of days between this day and another
     day
     @param other the other day
     @return the number of days that this day is away from
     the other (>0 if this day comes later)
  */
  int daysFrom(Day other) const;

  bool comesBefore (Day other) const;
  bool sameDay (Day other) const;

private:
  int daysSinceStart;  // start is 10/15/1582

  /* alternate implem:
  int theYear;
  int theMonth;
  int theDate;
  */
};
```

For example, if we were to look through proposed applications of the Day class and find designs with pseudocode like:

```
if (d is last day in its month)
   payday = d;
```

we would be happy with the ability of our Day interface to support this.

. . . .

Is Day complete? (2)

On the other hand, if we encountered a design like this:

```
while (payday falls on the weekend)
   move payday back one day
end while
```

we might want to consider adding a function to the Day class to get the day of the week.

. . . .

6.2.2 Are there redundant functions or functions that can be generalized?

Are there redundant functions or functions that can be generalized?

- Avoid unneded redundancies that make your class larger and provide additional code to maintain.
- Generalize when possible to provide more options to the application.

. . . .

Example: Day output

```
class Day
{
public:
  ⋮
  void print() const;
private:
  ⋮
};
```

Future applications may need to send their output to different places. So, it makes sense to make the print destination a parameter:

```
void print (std::ostream& out);
```

. . . .

Day output op

Of course, most C++ programmers are used to doing output this way:

```
cout << variable;
```

rather than

```
variable.print (cout);
```

So we would do better to add the operator. . .

. . . .

Day output op

```
class Day
{
public:
  ⋮
  void print(std::ostream out) const;
private:
  ⋮
};

inline
std::ostream& operator<< (std::ostream& out, Day day)
{
  dat.print (out);
  return out;
}
```

. . . .

6.2.3 Have you used names that are meaningful in the application domain?

Have you used names that are meaningful in the application domain?
An important part of this question is the "*in the application domain*".

- Good variable names should make sense to anyone who understands the application domain,
- even if they don't understand (yet) how your program works.

. . . .

Example: Book identifiers

- `getTitle`, `putTitle`, or `getNumberOfAuthors` are all fine.
- Not everyone who has worked with books would understand `getIdentifier`
- Better would be to recognize that suitable unique identifiers already exist

```
class Book {
public:
  ⋮
    std::string getISBN();
  ⋮
};
```

- the ISBN appears on the copyright page of every published book.

. . . .

6.2.4 Preconditions and Assertions

Are pre-conditions and assumptions well documented?

A *pre-condition* is a condition that the person calling a function must be sure is true, before the call, if he/she expects the function to do anything reasonable.

- Pre-conditions must be documented because they are an obligation upon the caller
  - And callers can't fulfill that obligation if they don't know about it

. . . .

Example: Day Constructor

```
class Day
{
   /**
      Represents a day with a given year, month, and day
      of the Gregorian calendar. The Gregorian calendar
      replaced the Julian calendar beginning on
      October 15, 1582
   */

public:

  Day(int aYear, int aMonth, int aDate);

   /**
      Returns the year of this day
      @return the year
   */
  int getYear() const;



  /**
     Returns the month of this day
     @return the month
  */
  int getMonth() const;

  /**
     Returns the day of the month of this day
     @return the day of the month
  */
  int getDate() const;



  /**
     Returns a day that is a certain number of days away from
     this day
     @param n the number of days, can be negative
```

```
    @return a day that is n days away from this one
  */
  Day addDays(int n) const;


  /**
    Returns the number of days between this day and another
    day
    @param other the other day
    @return the number of days that this day is away from
    the other (>0 if this day comes later)
  */
  int daysFrom(Day other) const;

  bool comesBefore (Day other) const;
  bool sameDay (Day other) const;

private:
  int daysSinceStart;  // start is 10/15/1582

  /* alternate implem:
  int theYear;
  int theMonth;
  int theDate;
  */
};
```

What pre-condition would you impose upon the *Day* constructor?

- A fairly obvious requirement is for the month and day to be valid.

```
  Day(int aYear, int aMonth, int aDate);
```

```
//pre: (aMonth > 0 && aMonth <= 12)
//   && (aDate > 0 && aDate <= daysInMonth(aMonth, aYear))
```

- All pre-conditions are, by definition, boolean expressions
- As we will see shortly, there's a significant advantage to writing them as proper C++ expressions

....

Example: Day Constructor (cont.)

This comment

```
class Day
{
   /**
      Represents a day with a given year, month, and day
      of the Gregorian calendar. The Gregorian calendar
      replaced the Julian calendar beginning on
      October 15, 1582
   */
   ⋮
```

suggests a more rigorous pre-condition

```
  Day(int aYear, int aMonth, int aDate);
  //pre: (aMonth > 0 && aMonth <= 12)
  //   && (aDate > 0 && aDate <= daysInMonth(aMonth, aYear))
  //   && (aYear > 1582 || (aYear == 1582 && aMonth > 10)
  //         || (aYear == 1582 && aMonth == 10 && aDate >= 15)
```

....

Example: MailingList getContact

```
#ifndef MAILINGLIST_H
#define MAILINGLIST_H

#include <iostream>
#include <string>

#include "contact.h"

/**
   A collection of names and addresses
*/
class MailingList
{
public:
  MailingList();
  MailingList(const MailingList&);
  ~MailingList();

  const MailingList& operator= (const MailingList&);

  // Add a new contact to the list
  void addContact (const Contact& contact);

  // Remove one matching contact
  void removeContact (const Contact&);
  void removeContact (const Name&);

  // Find and retrieve contacts
  bool contains (const Name& name) const;
  Contact& getContact (const Name& name) const;
  // combine two mailing lists
  void merge (const MailingList& otherList);

  // How many contacts in list?
  int size() const;


  bool operator== (const MailingList& right) const;
  bool operator< (const MailingList& right) const;

private:

  struct ML_Node {
    Contact contact;
    ML_Node* next;

    ML_Node (const Contact& c, ML_Node* nxt)
      : contact(c), next(nxt)
    {}
  };

  int theSize;
  ML_Node* first;
  ML_Node* last;

  // helper functions
  void clear();
  void remove (ML_Node* previous, ML_Node* current);

  friend std::ostream& operator<< (std::ostream& out, const MailingList& addr);
```

```
};

// print list, sorted by Contact
std::ostream& operator<< (std::ostream& out, const MailingList& list);


#endif
```

What pre-condition, if any, would you write for the `getContact` function?

. . . .

Have you used `assert` to “guard” the inputs?

An `assert` statement takes a single argument,

- a boolean condition that we believe should be true unless someone somewhere has made a programming mistake.
- It aborts program execution if that condition evaluates as false.
  - Can be “turned off” in release versions by a simple compiler switch
    - Though that might not be a good thing. . .

. . . .

Example: Guarding the Day Constructor

```
#include "day.h"
#include <cassert>

using namespace std;

Day::Day(int aYear, int aMonth, int aDate)
//pre: (aMonth > 0 && aMonth <= 12)
//   && (aDate > 0 && aDate <= daysInMonth(aMonth,aYear))
//   && (aYear > 1582 || (aYear == 1582 && aMonth > 10)
//        || (aYear == 1582 && aMonth == 10 && aDate >= 15)
{
  assert (aMonth > 0 && aMonth <= 12);
  assert (aDate > 0 && aDate <= 31);
  assert (aYear > 1582 || (aYear == 1582 && aMonth > 10)
          || (aYear == 1582 && aMonth == 10 && aDate >= 15));
  daysSinceStart = ...
}
```

. . . .

Example: guarding getContact

```
#include "mailinglist.h"
#include <cassert>

using namespace std;


Contact& MailingList::getContact (const Name& name) const
{
  ML_Node* current = first;
  while (current != NULL
     && name > current->contact.getName())
    {
      previous = current;
      current = current->next;
    }
  assert (current != NULL
     && name == current->contact.getName());
  return current->contact;
}
    ⋮
```

. . . .

Do Assertions Reduce Robustness?

Why do

```
assert (current != NULL
          && name == current->contact.getName());
```

instead of making the code take corrective action? E.g.,

```
if (current != NULL
     && name == current->contact.getName())
    return current->contact;
else
    return Contact();
```

. . . .

Do Assertions Reduce Robustness?

- Many people argue in favor of making code “tolerate” errors.
  - The ability to recover from errors is called *robustness*.
  - Robust handling of human interface errors is a good thing
- However, tolerance of pre-conditions is reckless.
  - A pre-condition violation is evidence of a bug in the *application*.
  - If the application is under test, it’s self-defeating to hide mistakes.
  - If application has been released, hiding bugs can lead to plusible but incorrect output, corrupted files & databases, etc.

. . . .

6.2.5 Are the data members private?

Are the data members private?

As discussed earlier, we strongly favor the use of encapsulated `private` data to provide information hiding in ADT implementations.

- E.g., permitting alternate implementations of the *Day* class

. . . .

Providing Access to Attributes

Two common styles in C++:

```
class Contact {
public:
  Contact (Name nm, Address addr);

  const Name& name() const {return theName;}
  Name&       name()       {return theName;}

  const Address& getAddress() const {return theAddress;}
  Address&       getAddress()       {return theAddress;}
      ⋮
```

- `getAttr`, `setAttr` as used for the Address attribute
- Attribute reference functions

. . . .

Attribute Reference Functions

```
class Contact {
    ⋮
  const Address& getAddress() const {return theAddress;}
  Address&       getAddress()       {return theAddress;}
```

Attribute reference functions can be used to both access and assign to attributes

```
void foo (Contact& c1, const Contact& c2)
{
  c1.name() = c2.name();
}
```

but may offer less flexibility to the ADT implementor.

- Consequently, not used as often as get/set

. . . .

6.2.6 Does every constructor initialize every data member?

Does every constructor initialize every data member?

Simple enough to check, but can prevent some very difficult-to-catch errors.

. . . .

Example: MailingList

Check this...

```
class MailingList
{
   ⋮
private:

  struct ML_Node {
    Contact contact;
    ML_Node* next;
    ML_Node (const Contact& c, ML_Node* nxt)
      : contact(c), next(nxt)
    {}
  };

  int theSize;
  ML_Node* first;
  ML_Node* last;
};
```

against this:

```
#include <cassert>
#include <iostream>
#include <string>
#include <utility>

#include "mailinglist.h"


using namespace std;
using namespace rel_ops;



MailingList::MailingList()
  : first(NULL), last(NULL), theSize(0)
{}

MailingList::MailingList(const MailingList& ml)
  : first(NULL), last(NULL), theSize(0)
{
  for (ML_Node* current = ml.first; current != NULL;
       current = current->next)
    addContact(current->contact);
}


MailingList::~MailingList()
{
  clear();
}

const MailingList& MailingList::operator= (const MailingList& ml)
{
  if (this != &ml)
    {
      clear();
      for (ML_Node* current = ml.first; current != NULL;
       current = current->next)
    addContact(current->contact);
    }
  return *this;
}


// Add a new contact to the list
void MailingList::addContact (const Contact& contact)
{
  if (first == NULL)
    { // add to empty list
      first = last = new ML_Node(contact, NULL);
      theSize = 1;
    }
  else if (contact > last->contact)
    { // add to end of non-empty list
      last->next = new ML_Node(contact, NULL);
      last = last->next;
      ++theSize;
    }
  else if (contact < first->contact)
    { // add to front of non-empty list
      first = new ML_Node(contact, first);
      ++theSize;
    }
  else
    { // search for place to insert
      ML_Node* previous = first;
      ML_Node* current = first->next;
      assert (current != NULL);
      while (contact < current->contact)
    {
      previous = current;
      current = current->next;
      assert (current != NULL);
    }
      previous->next = new ML_Node(contact, current);
      ++theSize;
    }
}


// Remove one matching contact
void MailingList::removeContact (const Contact& contact)
```

```
{
  ML_Node* previous = NULL;
  ML_Node* current = first;
  while (current != NULL && contact > current->contact)
    {
      previous = current;
      current = current->next;
    }
  if (current != NULL && contact == current->contact)
    remove (previous, current);
}


void MailingList::removeContact (const Name& name)
{
  ML_Node* previous = NULL;
  ML_Node* current = first;
  while (current != NULL
     && name > current->contact.getName())
    {
      previous = current;
      current = current->next;
    }
  if (current != NULL
     && name == current->contact.getName())
    remove (previous, current);
}



// Find and retrieve contacts
```

```
bool MailingList::contains (const Name& name) const
{
  ML_Node* current = first;
  while (current != NULL
     && name > current->contact.getName())
    {
      previous = current;
      current = current->next;
    }
  return (current != NULL
      && name == current->contact.getName());
}


Contact MailingList::getContact (const Name& name) const
{
  ML_Node* current = first;
  while (current != NULL
     && name > current->contact.getName())
    {
      previous = current;
      current = current->next;
    }
  if (current != NULL
      && name == current->contact.getName())
    return current->contact;
  else
    return Contact();
}
```

```
// combine two mailing lists
void MailingList::merge (const MailingList& anotherList)
{
  // For a quick merge, we will loop around, checking the
  // first item in each list, and always copying the smaller
  // of the two items into result
  MailingList result;
  ML_Node* thisList = first;
  const ML_Node* otherList = anotherList.first;
  while (thisList != NULL and otherList != NULL)
    {
      if (thisList->contact < otherList->contact)
    {
      result.addContact(thisList->contact);
      thisList = thisList->next;
    }
      else
    {
      result.addContact(otherList->contact);
      otherList = otherList->next;
    }
    }
  // Now, one of the two lists has been entirely copied.
  // The other might still have stuff to copy. So we just copy
  // any remaining items from the two lists. Note that one of these
  // two loops will execute zero times.
  while (thisList != NULL)
    {
      result.addContact(thisList->contact);
      thisList = thisList->next;
    }
  while (otherList != NULL)
    {
      result.addContact(otherList->contact);
      otherList = otherList->next;
    }
  // Now result contains the merged list. Transfer that into this list.
  clear();
  first = result.first;
  last = result.last;
  theSize = result.theSize;
  result.first = result.last = NULL;
  result.theSize = 0;
}

// How many contacts in list?
int MailingList::size() const
{
  return theSize;
}


bool MailingList::operator== (const MailingList& right) const
{
  if (theSize != right.theSize) // (easy test first!)
    return false;
  else
    {
      const ML_Node* thisList = first;
      const ML_Node* otherList = right.first;
      while (thisList != NULL)
    {
      if (thisList->contact != otherList->contact)
        return false;
      thisList = thisList->next;
      otherList = otherList->next;
    }
      return true;
    }
}


bool MailingList::operator< (const MailingList& right) const
{
  if (theSize < right.theSize)
    return true;
  else
    {
      const ML_Node* thisList = first;
      const ML_Node* otherList = right.first;
      while (thisList != NULL)
    {
      if (thisList->contact < otherList->contact)
        return true;
      else if (thisList->contact > otherList->contact)
        return false;
      thisList = thisList->next;
      otherList = otherList->next;
    }
      return false;
    }
}


// helper functions
void MailingList::clear()
{
  ML_Node* current = first;
  while (current != NULL)
    {
      ML_Node* next = current->next;
      delete current;
      current = next;
    }
  first = last = NULL;
  theSize = 0;
}


void MailingList::remove (MailingList::ML_Node* previous,
       MailingList::ML_Node* current)
{
  if (previous == NULL)
    { // remove front of list
      first = current->next;
      if (last == current)
    last = NULL;
      delete current;
    }
  else if (current == last)
    { // remove end of list
```

```
      last = previous;
      last->next = NULL;
      delete current;
    }
  else
    { // remove interior node
      previous->next = current->next;
      delete current;
    }
  --theSize;
}


// print list, sorted by Contact
std::ostream& operator<< (std::ostream& out, const MailingList& list)
{
  MailingList::ML_Node* current = list.first;
  while (current != NULL)
    {
      out << current->contact << "n";
      current = current->next;
    }
  out << flush;
  return out;
}


book1.setTitle(''bogus title'');
assert (book1.getTitle() == ''bogus title'');

book2 = book1;
```

```
assert (book1 == book2);
book1.setTitle(''bogus title 2'');
assert (! (book1  == book2));




catalog.add(book1);
assert (catalog.firstBook() == book1);>




string s1, s2;
cin >> s1 >> s2;
if (s1 < s2)       ''abc'' < ''def''
                       ''abc'' < ''abcd''


    x y

Exactly one of the following is true for any x and y
    x == y
    x < y
    y < x

 namespace std{

   namespace relops {
template <class T>
bool operator!= (T left, T right)
{
  return !(left == right);
}
```

```
template <class T>
bool operator> (T left, T right)
{
  return (right < left);
}



     using namespace std::relops;
```

. . . .

6.2.7 Have you appropriately treated the default constructor?

Have you appropriately treated the default constructor?

Remember that the default constructor is a constructor that can be called with no arguments.

Your options are:

1. The compiler-generated version is acceptable.
2. Write your own
3. No default constructor is appropriate
4. (very rare) If you don't want to allow other code to construct objects of your ADT type at all, declare a constructor and make it `private`.

. . . .

6.2.8 Have you appropriately treated the "Big 3"?

Have you appropriately treated the "Big 3"?

- Recall that the Big 3 in C++ class design are the copy constructor, the assignment operator, the destructor.
  - For each of these, if you do not provide them, the compiler generates a version for you.
- The *Rule of the Big 3* states that,

  if you provide your own version of any one of the big 3, you should provide your own version of all 3.

. . . .

Handling the Big 3

So your choices as a class designer come down to:

1. The compiler-generated version is acceptable for all three.
2. You have provided your own version of all three.
3. You don't want to allow copying of this ADT's objects
   - Provide private versions of the copy constructor and assignment operator so the compiler won't provide public ones, but no one can use them.

. . . .

The Compiler-Generated Versions are wrong when...

Copy constructor Shallow-copy is inappropriate for your ADT

Assignment operator Shallow-copy is inappropriate for your ADT

Destructor Your ADT holds resources that need to be released when no longer needed

. . . .

The Compiler-Generated Versions are wrong when... (2)

Generally this occurs when

- Your ADT has pointers among its data members, and
- You don't want to share the objects being pointed to.

. . . .

The Rule of the Big 3

The *Rule of the Big 3* states that,

if you provide your own version of any one of the big 3, you should provide your own version of all 3.

Why? Because we don't trust the compiler-generated...

- ...copy constructor if our data members include pointers to data we don't share
- ...assignment operator if our data members include pointers to data we don't share
- ...destructor if our data members include pointers to data we don't share

So if we don't trust one, we don't trust any of them.

. . . .

6.2.9 Does your assignment operator handle self-assignment?

Does your assignment operator handle self-assignment?

If we assign something to itself:

```
x = x;
```

we normally expect that nothing really happens.

But when we are writing our own assignment operators, that's not always the case. Sometimes assignment of an object to itself is a nasty special case that breaks thing badly.

. . . .

6.2.10 Does your class provide == and < operators?

Does your class provide == and < operators?

- *The compiler never generates these implicitly*, so if we want them, we have to supply them.
- The == and < are often required if you want to put your objects inside other data structures.
  - That's enough reason to provide them whenever practical.
- Also heavily used in test drivers

. . . .

6.2.11 Does your class provide an output routine?

Does your class provide an output routine?

- Even if not required by the application, useful (essential?) in testing and debugging.
  - Again, that's enough reason to always provide one if at all practical to do so

. . . .

6.2.12 Is your class const-correct?

Is your class const-correct?
In C++, we use the keyword `const` to declare constants. But it also has two other important uses:

1. indicating what formal parameters a function will look at, but promises not to change
2. indicating which member functions don't change the object they are applied to

These last two uses are important for a number of reasons

- This information often helps make it easier for programmers to understand the expected behavior of a function.

- The compiler may be able to use this information to generate more efficient code.
- This information allows the compiler to detect many potential programming mistakes.

. . . .

Const Correctness

A class is *const-correct* if

1. Any formal function parameter that will not be changed by the function is passed by copy or as a const reference (`const &`).
2. Every member function that does not alter the object it's applied to is declared as a const member.

. . . .

Example: Contact

Passed by copy, passed by const ref, & const member functions

```
#ifndef CONTACT_H
#define CONTACT_H

#include <iostream>
#include <string>

#include "name.h"
#include "address.h"

class Contact {
  Name theName;
  Address theAddress;

public:
  Contact (Name nm, Address addr)
    : theName(nm), theAddress(addr)
  {}


  Name getName() const   {return theName;}
  void setName (Name nm) {theName= nm;}

  Address getAddress() const     {return theAddress;}
  void setAddress (Address addr) {theAddress = addr;}

  bool operator== (const Contact& right) const
  {
    return theName == right.theName
      && theAddress == right.theAddress;
  }

  bool operator< (const Contact& right) const
  {
    return (theName < right.theName)
      || (theName == right.theName
      && theAddress < right.theAddress);
  }
};

inline
std::ostream& operator<< (std::ostream& out, const Contact& c)
{
  out << c.getName() << " @ " << c.getAddress();
  return out;
}
```

```
#endif
```

. . . .

Const Correctness Prevents Errors

- This code will not compile:

  ```
  void foo (Contact& conOut, const Contact& conIn)
  {
    conIn = conOut;
  }
  ```

- nor will this:

  ```
  void bar (Contact& conOut, const Contact& conIn)
  {
    conIn.setName (conOut.getName());;
  }
  ```

- The error messages aren't always easy to understand ("discards qualifier", "no match for. . . ")
  - Resist the temptation to strip away the "const"s or use type-casting to bypass the errors.

. . . .

Example: MailingList

Passed by copy, passed by const ref, & const member functions

```
#ifndef MAILINGLIST_H
#define MAILINGLIST_H
```

```
#include <iostream>
#include <string>

#include "contact.h"

/**
   A collection of names and addresses
*/
class MailingList
{
public:
  MailingList();
  MailingList(const MailingList&);
  ~MailingList();

  const MailingList& operator= (const MailingList&);

  // Add a new contact to the list
  void addContact (const Contact& contact);

  // Remove one matching contact
  void removeContact (const Contact&);
  void removeContact (const Name&);

  // Find and retrieve contacts
  bool contains (const Name& name) const;
  Contact getContact (const Name& name) const;


  // combine two mailing lists
  void merge (const MailingList& otherList);

  // How many contacts in list?
  int size() const;


  bool operator== (const MailingList& right) const;
  bool operator< (const MailingList& right) const;

private:

  struct ML_Node {
    Contact contact;
    ML_Node* next;

    ML_Node (const Contact& c, ML_Node* nxt)
      : contact(c), next(nxt)
    {}
  };

  int theSize;
  ML_Node* first;
  ML_Node* last;

  // helper functions
  void clear();
  void remove (ML_Node* previous, ML_Node* current);

  friend std::ostream& operator<< (std::ostream& out, const MailingList& addr);
};
```

```
// print list, sorted by Contact
std::ostream& operator<< (std::ostream& out, const MailingList& list);


#endif
```

....

6.3 Summary

Summary

This is a *checklist*

- Use it whenever you have the responsibility of designing an ADT interface
- These are not absolute rules, but are things that you need to think about
  - Violate them if necessary, but only after careful consideration

....

Part III

OO Analysis & Design: Workflows, Models, & Classification

Chapter 7

Workflows

Workflows

Certain activities seem to occur almost continuously. The *workflows* describe the "daily work" of a software designer. The four workflows are

- Analysis
- Design
- Implementation
- Validation

. . . .

7.1 What Happens In Each Workflow?

The Topic of Interest

The *topic of interest* (TOI) is the part of the system that you are working with at the moment.

- might be the entire system or a single ADT or even a single function
- depending upon where you are in the overall process

All the workflows have to be viewed in terms of the TOI.

. . . .

All the workflows have to be viewed in terms of a part of the system that you are working with at the moment. That "part" might be the entire system or a single ADT or even a single function, depending upon where you are in the overall process. (More on that idea shortly when we look at software development process models.) We'll call that part of the system the *topic of interest* (TOI). The TOI could be the entire system or a single ADT or even a single function within an ADT. For example, in the early stages of design you are concerned with the entire system, so the TOI is the entire system. As you begin to decompose the system into smaller parts, you begin to focus your attention on those parts one at a time. Each time you do this, your TOI becomes progressively more narrow. On the other hand, when you are testing the system, you may start with unit testing where your TOI is the development of tests for a single ADT. Later your focus will broaden, as during integration testing your TOI will be a subsystem, and during system testing the TOI expands to comprise (testing of) the entire system.

Whatever the TOI is, at any moment in time, software developers can expect to engage in the following 4 generic activities:

What Happens in Each Workflow?

- *Analysis* seeks to describe *what* is (or should be) going on in the topic of interest.

  In analysis, we consider the "world" surrounding the TOI and try to learn how the TOI must interact with that world.

  The focus of analysis is external, and there is an emphasis on interfaces.

. . . .

What Happens in Each Workflow?

- *Design* considers the questions of *how* our TOI will accomplish the goals established in the most recent round of analysis.

  The focus of design spans the external and the internal.

- *Implementation* is the rendering of the design decisions into a final form.

  "Final" in this case refers only to this round of work on the TOI.

- *Validation* consists of any activities we employ to determine whether the implementation is acceptable and whether it is time to move on to a different TOI.

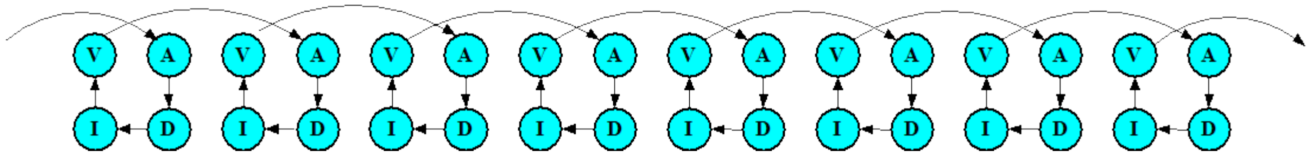
  Validation can take many forms, including testing, reviews with other team members or with the domain experts, and checks for completeness and internal consistency.

. . . .

7.2 Iterative Application of the Workflows

Iterative Application of the Workflows

- At the low level, a continuous cycle through the four workflows
- The TOI changes, but the daily activities represented by the 4 workflows remains the same.



. . At the low level, we should consider the total process of developing a software system as a continuous cycle through the four workflows, analysis, design, implementation, and validation. The TOI changes, but the daily activities represented by the 4 workflows remains the same.

7.3 Examples

Examples

A few examples of a possible single "turn" through the ADIV cycle follow.

. . . .

7.3.1 Example: Class Discovery

Example: Class Discovery

Topic of interest:

What are the basic characteristics of a book in the library?

. . . .

Analysis:

- Many of the characteristics are quite obvious. Books have titles, authors (possibly multiple), publishers, ISBN numbers, etc.
- However, someone looking through the library's card catalog notes that many of the books are labeled "copy 1", "copy 2", etc.
  - Based on this observation, the team got into a discussion of whether the term "book" actually refers to the thing on the shelf or to something more abstract. Is "War and Peace" a book even if the library does not own a copy? If the library has two copies, what do we call the common "thing" that these are copies of?

. . . .

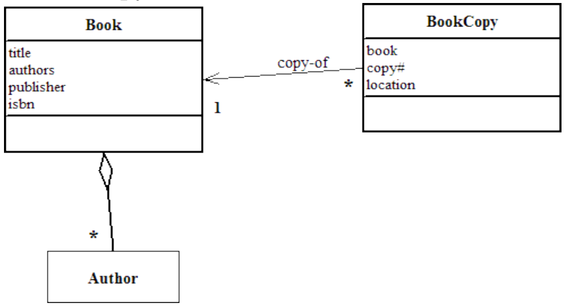
Design:

- The team proposed that the term "book" be used for the abstract idea of the thing that an author writes.
- The term "copy" or perhaps "bookcopy" will be used for a physical copy of a book.
- Books have all the attributes previously described.
- Book copies have attributes of location, copy number, and, of course, the book that they are a copy of.

. . . .

Implementation:

The team draws a simple UML class relationship diagram (these diagrams are introduced in a later lesson) describing the Book, BookCopy, and Author classes:



. . . .

Validation:

- The team discusses with a librarian their decision about proper terminology and about the key attributes.
- The librarian suggests that the distinction between "book" and "copy of a book" is technically correct, but often muddled in common use.
- The librarian notes that books have other important attributes, including date of publication and subject area. Finally, the librarian suggests that the focus on books may be too narrow, as the library actually houses many different kinds of publications.

. . . .

7.3.2 Example: ADT Interface

Example: ADT Interface

Topic of interest: Developing the ADT interface for a `Book`.

. . . .

Analysis:

The developer looks at the above diagram and notes that each book can have multiple authors, so the interface must provide some way to access an unknown number of author objects.

. . . .

Design:

The developer suggests using an iterator for this purpose, knowing this to be a common idiom in C++ for accessing an unknown number of "contained" objects.

. . . .

Implementation:

The developer adds the lines

```
typedef ... AuthorIterator;
```

and

```
AuthorIterator begin() const;
AuthorIterator end() const;
```

to the class declaration for `Book`.

. . . .

Validation:

The developer looks at some of the proposed algorithms that will work with books and authors, checking to see if those algorithms could be coded using an iterator style.

Satisfied that it will do so, the developer distributes the new class declaration to the rest of the development team for their approval.

. . . .

7.3.3 Example: ADT Implementation

Example: ADT Implementation

Topic of interest: Implementing the Book ADT

. . . .

Analysis:

- The programmer reviews the required attributes and behaviors of the book class and concludes that it is pretty straightforward, except for the question of how to store the authors.
- Reviewing the various application algorithms that manipulate books and authors, the programmer notes that adding and removing authors is fairly uncommon.
  - The programmer also notes that there are no instances of needing "random access" to the authors. The only algorithms that access the authors will process each author in turn.

. . . .

Design:

The programmer decides that the `std::list` would provide appropriate performance and would make for a simple implementation as well.

. . . .

Implementation:

The programmer writes the std list into the private section of the class declaration and codes the accompanying function bodies.

. . . .

Validation:

The programmer runs the previously developed self-checking unit test on the new `Book` implementation.

. . . .

7.3.4 Example: Training Manuals

Example: Training Manuals

Topic of interest: Preparing training documents for the new automated check-in system

. . . .

Analysis:

- The author reviews the (use-cases) scenarios for interactions between library staff, patrons, and the other library objects relating to check-in (return) of copies.
- The author makes a list of those that will require explanation to the library staff.

. . . .

Design:

The author decides which scenario variants and interactions are important and common enough to be needed in the initial training, and which can be relegated to reference documentation.

- The common and important items are arranged into related groups, and the author creates an overall outline that encompasses those groups as chapters/sections of the training manual.

. . . .

Implementation:

The author writes the relevant chapters and sections.

. . . .

Validation:

The sections are submitted to one of the system designers to be checked for accuracy and to one or more domain experts to be checked for understandability.

. . . .

Chapter 8

Software Development Processes

Software Process Models

The workflows are important, but they are too low-level to guide us through the entire lifetime of a development project. As we saw in the earlier examples, the ADIV cycle can apply to a wide variety of different activities. How do we know *when* to move from class discovery to ADT design, and when to start coding?

We need some sort of overall strategic plan to help us choose appropriate topics of interest for our ADIV cycles. That's the purpose of a software development process.

A *software development process* is a structured series of activities that comprise the way in which an organization develops software projects.

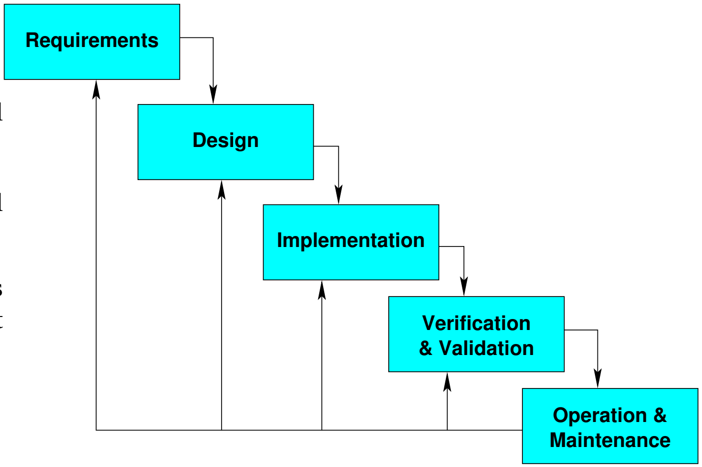
. . . .

8.1 The Waterfall Model

The Waterfall Model

Perhaps the best-known and most widely used process model is the *waterfall model*.

- Some of the names of the phases in this process model are the same as the names of some of our workflows.
- You need to rely on context to know whether "Design" is referring to a workflow or to a phase of the development process.



. . . .

The waterfall model gets its name from the fact that the first diagrams of this process illustrated it as a series of terraces over which a stream flowed, cascading down from one level to another. The point of this portrayal was that water always flows downhill - it can't reverse itself. Similarly, the defining characteristic of the waterfall model is the irreversible forward progress from phase to phase.

One thing that you may notice is that some of the names of the phases in this process model are the same as the names of some of our workflows. That's unfortunate, but we pretty much have to live with it. You need to rely on context to know whether "Design" is referring to a workflow or to a phase of the development process. That should not really be all that difficult. It should be obvious in most cases whether what is being discussed is a low-level daily activity or a major phrase of development that may span several months.

8.1.1 Requirements analysis

Requirements analysis

This phase establishes what the customer requires from a software system. The primary product of this phase is a requirements document.

What is a requirement?

May range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification

- This is inevitable as requirements may serve a dual function
  - May be the basis for a bid for a contract - therefore must be open to interpretation
  - May be the basis for the contract itself - therefore must be defined in detail
  - Both these statements may be called requirements

. . . .

Sample Reqts Specification
1.1 The user should be provided with facilities to define the type of external files
1.2 Each external file type may have an associated tool which may be applied to the file.
1.3 Each external file type may be represented as a specification from the user's display.
1.4 Facilities should be provided for the icon representing an external file type to be defined by the user.
1.5 When the user selects an icon representing an external file, the effect of that selection is to apply the tool associated with the type of the external file to the file represented by the selected icon.

. . . .

The requirements document

- The requirements document is the official statement of what is required of the system developers
  - Specify external system behaviour
  - Specify implementation constraints
  - Serve as reference tool for maintenance
- It is not a design document. As far as possible, it should indicate WHAT the system should do rather than HOW it should do it

. . . .

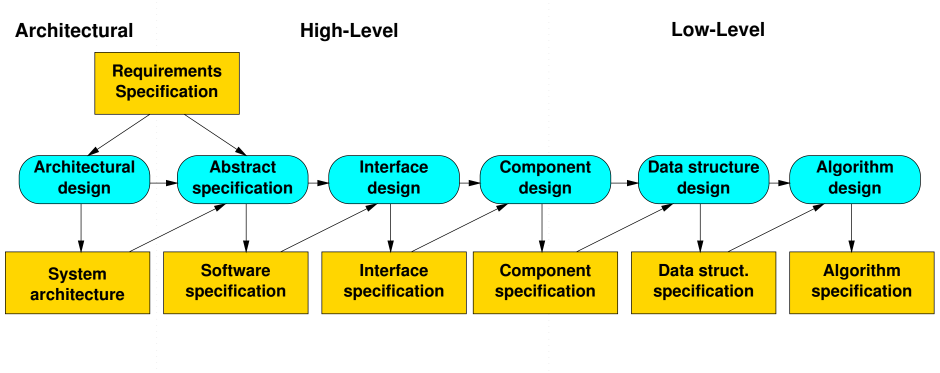
8.1.2 Design

Design

This phase seeks to derive a solution which satisfies the software requirements. The primary product of this phase is design documentation.

. . . .

Phases of design



Design can be broken down into three main phases

- *architectural design*, the collection of decisions that need to be common to all components.

  Examples of architectural design decisions would be

  – Will the system run on a single machine or be distributed over multiple CPUs?

  – Will outputs be stored in a file or in a database?

  – How will run-time errors be handled?

- *high-level* design, dividing the system into components, and

- *low-level* design (choosing the data structures and algorithms for a single component).

. . . .

Depending upon how formal a team's the overall process is, there may be several different design activities and corresponding design documents spread out over these three phases.

8.1.3 Implementation and unit testing

Implementation and unit testing

Writing the code - probably the most familiar activity.

The primary product of this phase is code - both application and test code.

. . . .

8.1.4 Verification and Validation

Verification and Validation

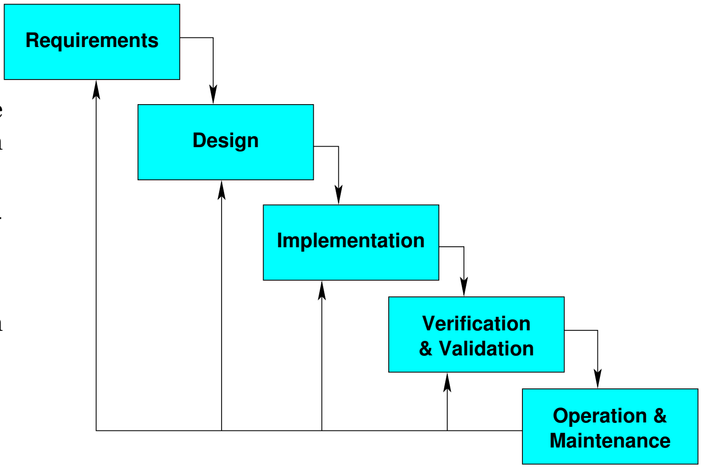
Verification & Validation: assuring that a software system meets the users' needs. The primary product of this phase is a test report.

The principle objectives are:

- The discovery of defects in a system
- The assessment of whether or not the system is usable in an operational situation.

. . . .

V & V



Although the waterfall model shows this as a separate phase near the end, we know that some forms of V&V occur much earlier.

• Requirements are validated in consultation with the customers.

• Unit testing occurs during Implementation, etc.
So this phase of the waterfall model really describes system and acceptance testing.

. . . .

8.1.5 Operation and maintenance

Operation and maintenance

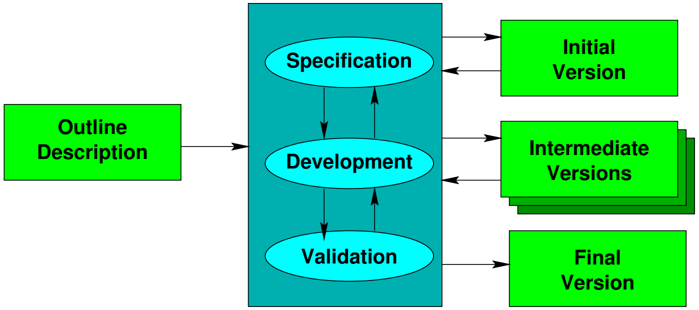
As requirements evolve and bug reports come in from the field, someone must

- prioritize changes
- make the changes
- validate the changes
  - new test cases
- validate that change does not break previously working code
  - regression testing

. . . .

8.2 Iterative/Incremental Development

Iterative/Incremental Development



. . . .

As a counter-reaction to what many believe to be an overly rigid waterfall model, there are a variety of incremental approaches that emphasize quick cycles of development, usually with earlier and more user-oriented validation.

Specification (analysis) and development are interleaved.

There is a greater emphasis on producing intermediate versions, each adding a small amount of additional functionality. Some of these are *releases*, either external (released outside the team) or internal (seen only by the team), which may have been planned earlier.

The evolutionary approach is conducive to exploration of alternatives. Some projects employ *throw-away prototyping*, versions whose code is only used to demonstrate and evaluate possibilities. This can lead to insight into poorly understood requirements.

Some waterfall projects may employ evolutionary schemes for parts of large systems (e.g., the user interface).

Risks of evolutionary schemes include poor process visibility (e.g., are we on schedule?), continual small drifts from the key architecture leading to poorly structured systems.

Incremental processes are periodically "rediscovered". For example, Extreme Programming is an evolutionary approach with heavy emphasis on involvement of the end-users in planning and on continuous validation.

Iterative Development

Advantages:

- conducive to exploration of alternatives
  - e.g., throw-away prototyping
  - can lead to insight into poorly understood requirements (e.g., GUIs)

Disadvantages:

- poor process visibility (e.g., are we on schedule?),
- continual small drifts from the key architecture leading to poorly structured systems.

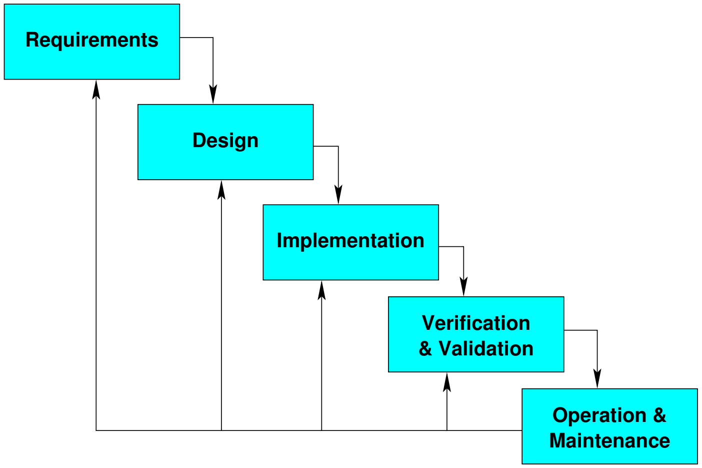
. . . .

8.3 General OO Analysis & Design

If we were to replace the word "Requirements" in the traditional waterfall model with "Analysis", then you might be tempted to view this as an embodiment of our ADIV workflow cycle. That's not entirely accidental. In fact, many variations of the waterfall diagram actually refer to the first phase as "requirements analysis", because analysis, in the sense that we have been describing it (determining *what* the system needs to do), is the primary activity underlying the construction of a requirements document.

Now, the ADIV cycle is a low-level workflow model, but this suggests that something rather similar can be used as a strategic project plan by making the "topic of interest" the entire system. The idea of doing OOAD is not at all inconsistent with the traditional waterfall.

However, OOAD brings its own nuance to how these activities get carried out.



General OO Analysis & Design

OOAD is largely viewed as a model-building activity. The primary models that we find are

- domain model
- analysis model
- design model

and OOAD emphasizes a smooth evolution from one to the other.

. . . .

8.3.1 Domain Models

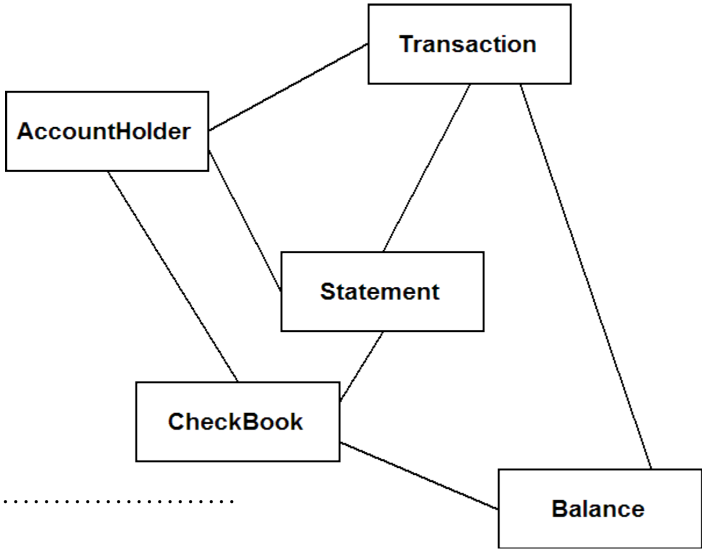
Domain Models

A *domain model* is a model of the application domain as it currently exists, *before* we began our new development project. The point of the domain model is to be sure the development team understands the world that the system will work in.

The domain model describes the world in terms of objects interacting with one another via messages. We'll see techniques for capturing this kind of idea shortly.

Not every project needs a domain model (e.g., If the team has done several projects already in this application domain)[1] In that case, the team may already have a good understanding of the domain.

Even if a document describing the domain model is desired, domain models tend to be highly reusable since the world around our software systems usually changes fairly slowly.



. . . .

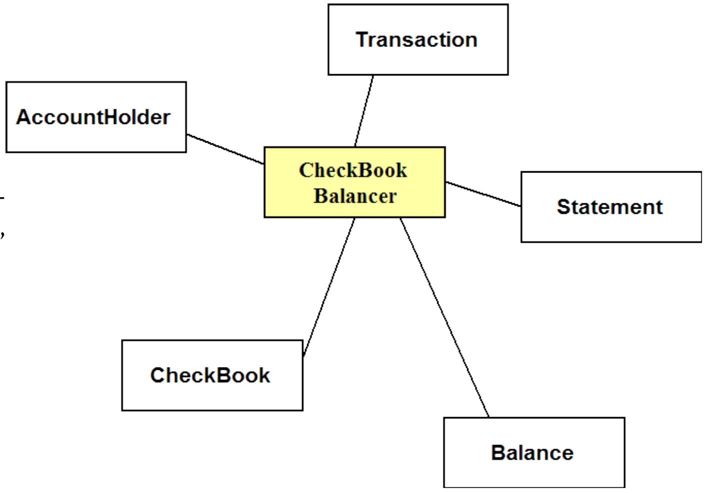
8.3.2 Analysis Models

Analysis Models

An *analysis model* is a model of how the world will interact with the software system that we envision. As such, it is our statement of just *what* the system will do when it is working.

[1] Fortunately, most companies work in a small number of application domains. If they hired you last month to develop software for analyzing seismic data for petroleum engineering, they are unlikely to ask you to develop a compiler or a word processor next month.

There is a real temptation to simply assume that the automated system will simply squat in the middle of the world, interacting with all the real world objects, sort of like this:
Or if you prefer,...

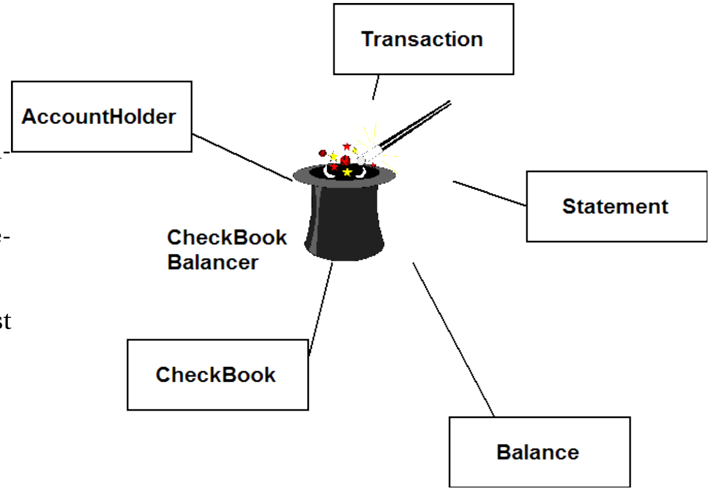


. . . .

It's Magic!

Poof! We have an analysis model!
Not wrong, per se, but it's certainly not helpful.
Such an approach is fundamentally at odds with the OO philosophy

- We should look to the real world to suggest how to decompose our system.
- In this case, we have not decomposed it at all – just wrapped it up into a single black box.
  - All the hard decisions still need to be made.

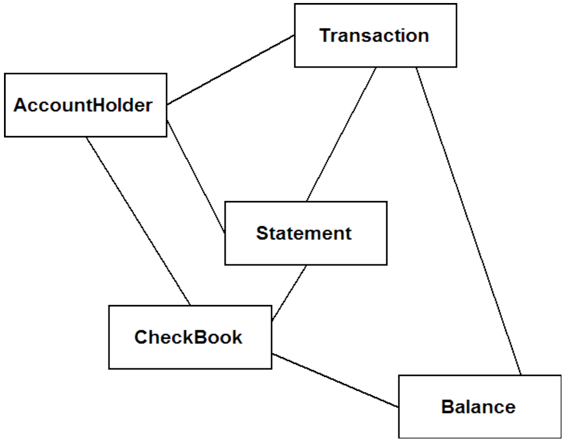


. . . .

In essence we have not done any analysis at all here. This "model" isn't *wrong*, per se, but it's certainly not helpful. We've basically thrown away everything we've learned in the domain model about how objects really interact. We're treating the new program as a simple box, with no knowledge of its internal structure, Essentially, we've just deferred all the hard questions to the upcoming design.

Evolving the Analysis Model

What we really hope for is an evolution from our domain model to our analysis model. The OO philosophy tells us that the classes and interactions of our domain model…



. . . should carry over into our analysis.

. . . .

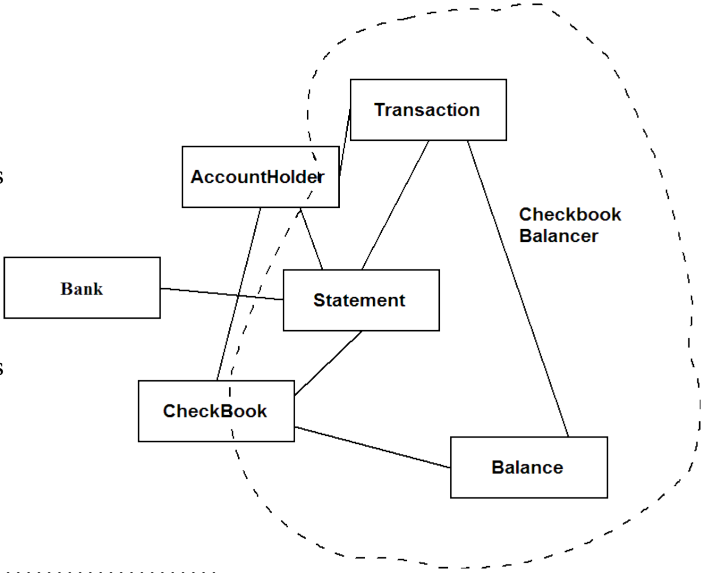
In essence, we hope to retain these classes, add more detail to our understanding of them, and to establish a boundary that tells us which of these classes and behaviors will be automated, which will remain entirely unautomated, and which will have some portion automated while other parts remain external.

The Boundary

...establish a boundary that tells us which of these classes and behaviors will

- be automated
- remain external
- be a mixture of the two

The system, then, remains a collection of interacting objects rather than an unstructured black box.



....

There's a definite overlap in the purpose of a requirements document and an analysis model. Some will regard the analysis model as a kind of requirements specification. In some projects, though, a requirements document will still be required as something for customers or management to sign off on. But the analysis model is the basis from which the eventual requirements document is derived.

8.3.3 Design Models

Design Models

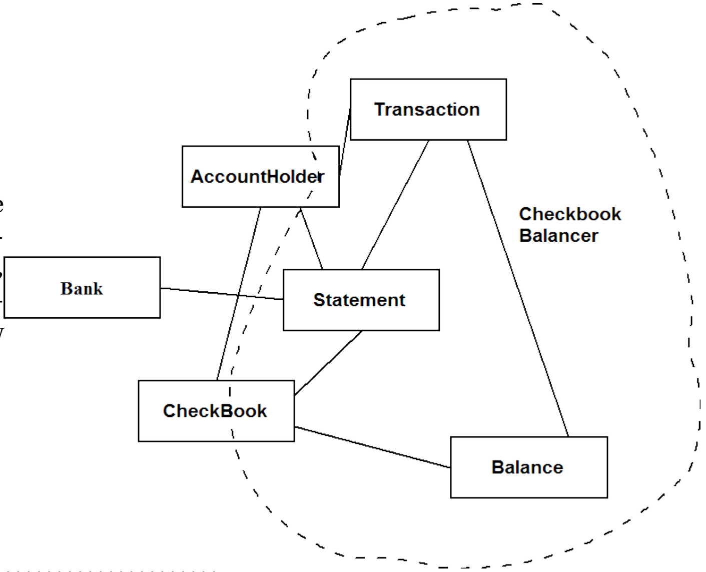
Design is concerned with *how* we can get the system to do the things the analysis model says it should do.

Designs are expressed via a *design model*,

which, like the earlier models, also is based upon the idea of interacting objects. In fact, if we believe in the OO philosophy, we would expect the objects and messages in the design model to bear a close resemblance to those from the analysis model. Hence we typically do not start building the design model from scratch so much as we *evolve* the analysis model into the design by adding detail, resolving conflicts, etc.

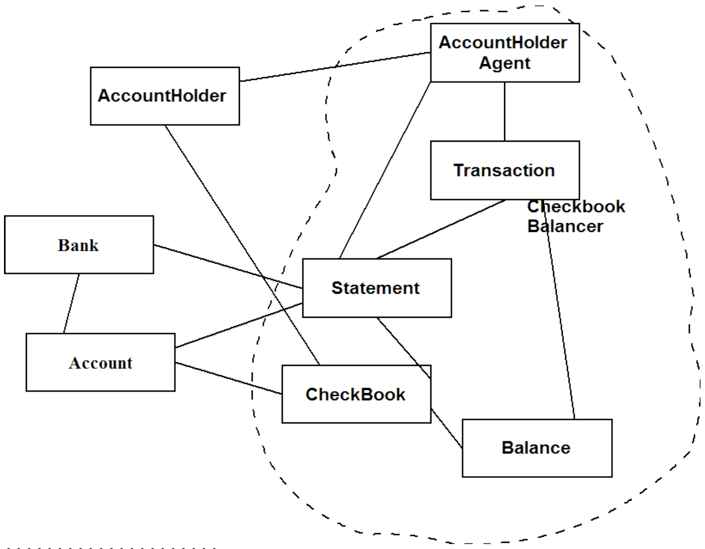
Design Model: Agents

A common step in OOD is to separate the classes that, in the analysis model, are partially implemented and partially external, into an purely external class representing the original, real-world objects, and an *agent* class that implements the automatable functions that will now be performed by the new system.

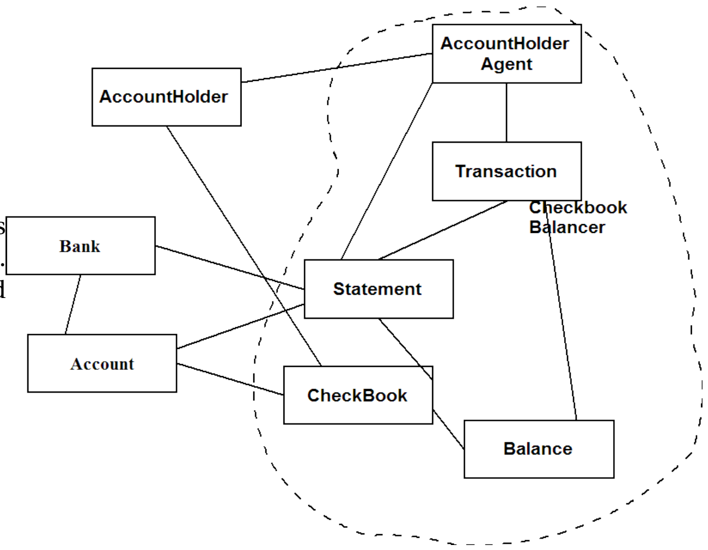


Design Model: Agents (2)

Here we have created an agent for AccountHolders



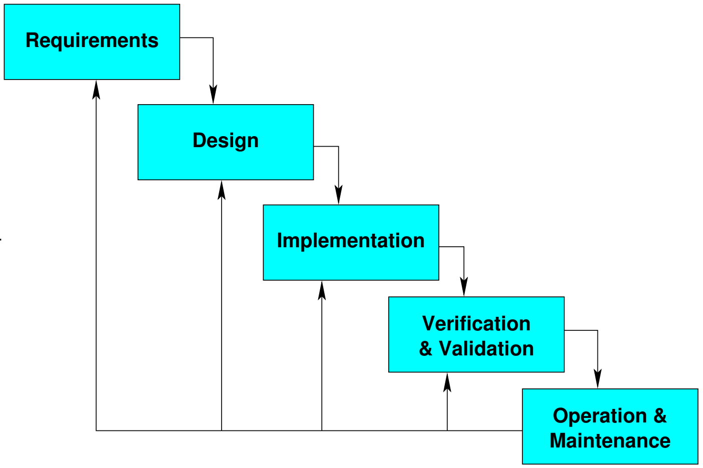
Design Model: Interfaces



Then we can look closely at the lines (interactions) that cross the border from the outside world into the automated system. What do we call an interaction between an external entity and an internal automated entity?

. . . .

Design Model: Interfaces



That's an *interface*, and so

• each of those crossing points is a portion of the user interface (or an interface with other automated systems).

. . . .

MVC

A common organizing principle for such interfaces is to distinguish between

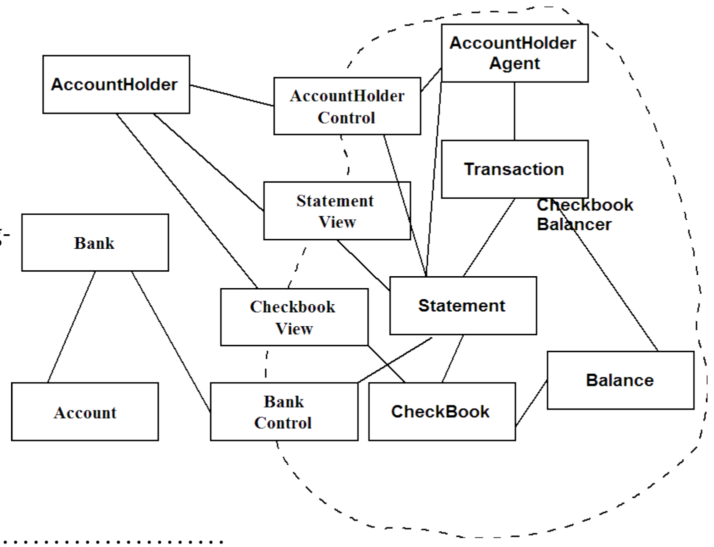
- *viewers* that portray the contents of automated classes and
- *controllers* that accept external signals (e.g., mouse clicks, keyboard data) and interpret those signals as instructions to automated classes and to viewers.
- Such viewers and controllers should be implemented as separate classes from the *model* classes that were derived from the analysis model.

. . . .

MVC Example

We then wind up with classes specifically devoted to managing the interface between the internal and external worlds.

- This is the Model-View-Controller pattern:



. . . .

Model-View-Controller

The MVC pattern establishes specific requirements so that GUI changes have minmal impact on the rest of the system.

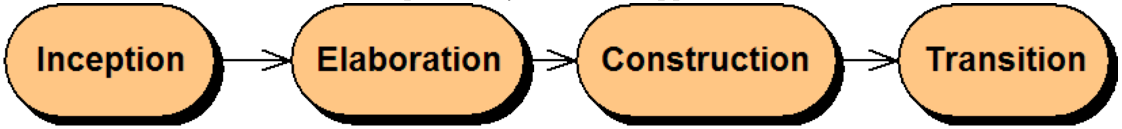
- The model classes must not depend on (make use of the interface of) the view and controlelr classes.
- The view classes must not depend on the controller classes

. . . .

8.4 The Unified Process Model

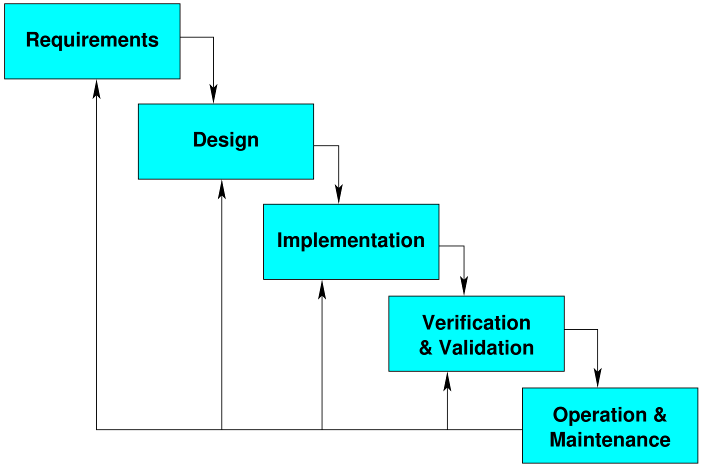
The Unified Process Model

For larger projects, we may want something a bit more formal than the general OOA&D process. The *Unified Process* was developed by Jacobsen, Booch, and Rumbaugh, who were already some of the biggest names in OOA&D before they decided to collaborate on a unified version of their previously distinctive approaches.



. . . .

Unified Model Phases



- Inception: initial concept
- Elaboration: exploring requirements
- Construction: building the software
- Transition: final packaging

. . . .

8.4.1 Inception

Inception

- Pitching the project concept
- Usually informal, low details.
  - "Perhaps we should build a . . . "

. . . .

8.4.2 Elaboration

Elaboration

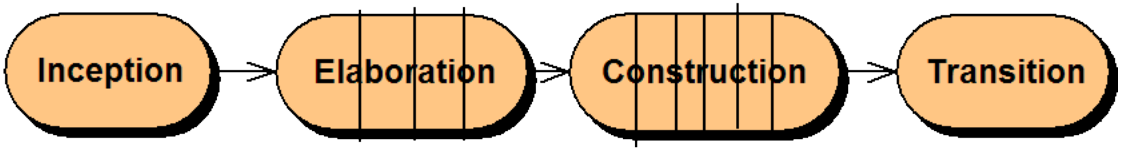
- Adding detail to our understanding of what the system should do.
- Produces
  - Domain model
  - Analysis model
  - Requirements document
  - Release plan

Among these products, only the release plan is new.

. . . .

Elaboration: Releases

In point of fact, each of the process phases may be divided into increments:



Such increments are most common in the elaboration and construction phases.

Each increment produces a *release* - some kind of product whose existence and/or acceptance by management shows that we are ready to move on.

A *release* represents the implementation of some part of the required functionality.

. . . .

Elaboration: Release Plans

The *release plan* records decisions about

- How many releases there will be
- What functionality will be added with each release
- When the releases will be made
- Which releases are internal (i.e., only the development team sees them) and which are external

. . . .

8.4.3 Construction

Construction

Perhaps the most familiar of the phases. This merges the waterfall activities of Design and Implementation.

- Each construction increment adds new functionality

- Documentation is constructed in addition to source code
- Each increment must be separately tested
- Each increment must be integrated and tested

. . . .

8.4.4 Transition

Transition

Activities that can't be done incrementally during construction, e.g.,

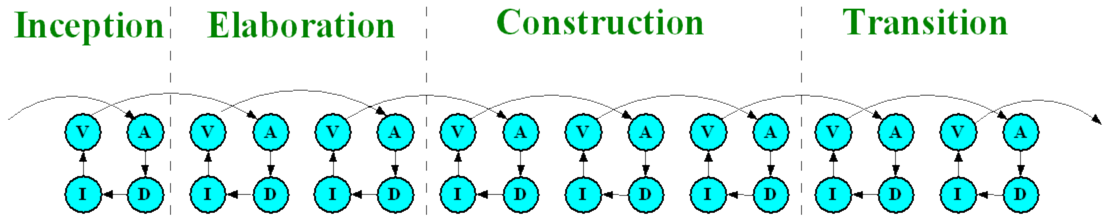
- performance tuning
- user training

. . . .

8.5 The Unified Process and Workflows

The Unified Process and Workflows

It should be noted again that the software development process model provides overall strategic guidance. The day-to-day "tactical" activity is still described by our four workflows.



. . . .

Chapter 9

Discovering and Documenting Classes

9.1 Classification

Classification

Classification: Where do Classes Come From?

A key step in any OO analysis or design is identifying the appropriate classes.

- In practice, this process is
  - incremental

    We tend to add a few classes at a time.
  - iterative

    We may revisit earlier decisions and change them.

    We often identify classes and then later add details about their relationships to other classes.

. . . .

Grouping Objects into Classes

We may identify groups of objects as a class because they have common

- properties,

  e.g., we regard all things that have titles, authors, and textual content as *documents*, regardless of whether they are in a print medium, a file, or even chiseled into a set of stone tablets.

  – Don't make the mistake of grouping things into a class because they have common property *values*.

    * A "collection of documents" can be a class. The values of that class can be collections that were selected by many different criteria.
    * By contrast, the collection of documents written by Mark Twain (i.e., whose *author* property has the value *Mark-Twain*) is not a class. It's just a particular value of the "collection of documents" class.

- behaviors,

  e.g., the set of all documents that can be loaded from and saved to a disk might represent a distinct class *ElectronicDocuments*.

....

9.2 Driving the classification process

Driving the classification process

Where do we get the information from which we can identify classes during analysis?
The "program as simulation" philosophy suggests that we should be looking for a model of the "real world".

- Initially we will build that model by looking at informal English descriptions of the world.

- Later, from use cases (scenarios)

. . . .

Our first pass is often informal, based on the documents at hand.

We will eventually formalize this process by writing, in conjunction with our domain experts, use-cases (scenarios) of sequences of actions in the world and analyzing those scenarios to see what they suggest about classes of objects in the world and the responsibilities of those classes.

Working from Informal Descriptions

Generally, this is done at the start of construction of a *domain model* or, if no domain model is needed, of the *analysis model*.

A fairly simple way to get started is to scan the informal statement looking for noun phrases and verb phrases

- Nouns represent candidate objects
- Verbs the messages/operations upon them (responsibilities)

This doesn't scale well to large projects/documents, but it is simple and often a useful starting point.

After that, we move on by exploiting our knowledge to assign the responsibilities to the appropriate and to classes, refine our choice of classes and responsibilities.

. . . .

Use-Case analysis

A *use-case* is a particular pattern of usage, a scenario that begins with some user of the system initiating a transaction or sequence of related events.

We analyze use-cases to discover the

- objects that participate in the scenario
- responsibilities of each object
- collaborations with other objects

More on this in later lessons.

. . . .

9.3 Informal Documentation: CRC Cards

CRC Cards

Early in our development process, we won't want to be slowed down by the need to construct detailed documentation that looks "pretty" enough to show people outside our team (management or domain experts).

CRC cards are a popular way of capturing early classification steps.

. . . .

CRC

CRC (Class, Responsibility, & Collaborators) cards are a useful tool during early analysis, especially during team discussions. They are not exactly a high-tech tool:

- 4x6 index cards
- used to take notes during analysis,
- as a concrete symbol for an object during disucssion
  - cards can be stacked, moved, etc. to illustrate proposed relationships

. . . .

A low-tech (no-tech?) approach is often useful in early brainstorming sessions. The index cards can serve as a concrete symbol for object during discussion. People trying to make a point may stack the cards, move them around, etc., while discussing proposed relationships.

CRC Cards are Informal Documentation

- The point of CRC cards is to capture info about an analysis discussion without slowing down that discussion so someone can take nicely formatted notes.
- They aren't pretty.
  - They aren't something you ever want to show your customers or even your own upper-management.

  - If you come out of a group meeting and your CRC cards aren't smudged, dog-eared, with lots of scratched-out bits, you probably weren't really trying.

. . . .

The point of CRC cards is to capture info about an analysis discussion without slowing down that discussion so someone can take nicely formatted notes. (Every now and then I see someone announce a new online tool for letting you type into a PC to write CRC cards that will then be printed out in nice neatly formatted output. That really misses the point. Ever been in a meeting where some single person was trying to type up all the important stuff being said? What does that usually do to the discussion dynamic?)

9.3.1 CRC Card Layout

CRC Card Layout

- labeled with class name
- divided into two columns
  - *responsibilities*
    A high-level description of a purpose of the class
    - attributes
    - behaviors
  - *collaborators* other classes with which this class must work with (send messages to) to fulfill this class's responsibilities

ClassName	
responsibility 1	
responsibility 2	collaborator 1
responsibility 3	collaborator 2

. . . .

CRC Card Example

Librarian	
Handles checkout and checkin of publications	Patrons
Reshelves books	Publications
Manages new acquisitions of publications	Inventory
Has name, ID#, branch assignment	Catalog

9.3.2 Assigning Responsibilities

Assigning Responsibilities

- The responsibilities will eventually evolve into messages that can be sent to this class and then into member functions of an ADT.
- Being aware of that intention can be a good indicator of which class should receive a particular responsibility.

Assigning Responsibilities Example

For example, if I were told "a library patron will give a librarian a book to be checked out", I would model this as

Librarian	
Handles checkout of books for patrons ⋮	

Patron	
	Librarian

but *not* as

Librarian	
⋮	

Patron	
Asks librarian to check out book	Librarian

When A does B to C

A useful rule of thumb is that if "A does B to C", then

- "doing B" is a responsibility.
- But it is *usually* a responsibility of class C, not of A.
- C is then a collaborator of A

Natural language being the flexible and imprecise tool that it is, there are many exceptions to this. But that's where thinking of responsibilities as future functions can help. In programming terms, statements like "A does B to C" often occur in context where we are describing a series of steps being enacted because someone or something else asked A to fulfill some higher-level responsibility. In pseudo-code terms, we might say

```
void A::fulfillSomeOtherResponsibilityOfA(C c1)
{
    ⋮
  c1.B();
    ⋮
}
```

in which case it is clear that B is a responsibility of class C, and C should be listed as a collaborator on A's card (along with that "some other responsibility").

. . . .

Containers

A variation on this rule of thumb occurs when managing collections.

If I told you that "the librarian adds the book's metadata[1] to the catalog", I would expect you to model that as

Catalog	
Permits addition of metadata	
⋮	

and not as

[1] *Metadata* refers to the set of properties that identify and describe a document or other collection of data. Typical metadata fields are author, title, date of publication, etc. "Metadata" is a perfect example of the type of specialized vocabulary that the people working in the Library World would understand and that you as a software developer assigned to that world would need to learn to use when communicating with them.

Metadata	
Can be added to a catalog ⋮	

. . . .

Containers (cont.)

Again, by analogy with programming, we understand that if you had something like

```
vector<int> v;
set<int> myList;
   ⋮
v.push_back(23);
myList.insert (23);
```

You would regard the ability to push data onto the end of a vector or to add data to a set as operations of the vector and the set, not as operations of the int data type.

You would never say:

```
23.insertInto (myList);
```

Not only is this ugly, but it suggests that we would need to predict all the possible container classes that will ever be written to hold integers, and to somehow add their specific add/insert/push operations to the set of permitted operations of the *int* type.

. . . .

Similarly, we'd expect that metadata could be added to many other kinds of containers as well as the library catalog. The ability to hold multiple instances of metadata is a responsibility of the container, not of the thing contained.

9.3.3 Attributes

Attributes

A common variation (used in the textbook) is to use the backs of the cards to list attributes.

We won't do that. We'll list known attributes among the responsibilities,

- By convention, "Has" introduces an attribute or list of attributes.

Librarian	
Handles checkout and checkin of publications	Patrons
Reshelves books	Publications
Manages new acquisitions of publications	Inventory
Has name, ID#, branch assignment	Catalog

Attribute Conventions

- Singular and plurals are used in noun phrases to distinguish between single occurrences ("name") and collections ("publications").
  - Avoid suggestions of specific data structures (e.g., "array of publications").
  - More generic terms “collection”, “sequence” (if ordered) are OK.

9.3.4 Empty Columns

Empty Columns

It’s OK for one or the other column to wind up being empty.

- An empty list of responsibilities is often associated with classes that are *actors* - they send messages to other classes but nothing in this world sends messages to them.

  These often correspond to external systems or people who are acting spontaneously to initiate some action.
- An empty list of collaborators can indicate that a class is a *server* - it accepts messages but sends out none. This is most likely in a "lower-level" class that is little more than a collection of attributes. Most systems have many such classes.
- If both columns remain empty, however, that’s a good sign that the class may not be needed in your model.
  - Be careful, though, about jumping to this conclusion too soon. Dropping classes form the mode should only be done when you are nearing the completion of the model.

9.3.5 Common Mistakes in CRC Cards

Common Mistakes in CRC Cards

- Premature design: You aren't doing code, selecting data structures, etc., yet. If you *were* doing those, you shouldn't be using CRC cards.
- Overly specific collaborators: There is no significance between the vertical match up of the responsibilities and the collaborators, There's no implication that a particular collaborator goes with a responsibility "on the same line".

  Consequently, there's no reason to list a collaborator twice.
- Mis-assigned responsibilities: already discussed
- Inconsistent collaboration: If you place a class B in the collaborator list of card A, then there needs to be some responsibility of B that it actually makes sense for A to call upon or make use of.

  In particular, if class B has an empty responsibilities column, it really can't appear as a collaborator of anything at all!

. . . .

Chapter 10

Example: Domain Model Using CRC Cards

Example of Early Analysis

This lesson works through an example of the early stages of analysis.

- Available documentation is natural language, fairly general in nature
  - Natural language is always tricky to work with. Ambiguities and contradictions are common.
  - You must read carefully and critically.
- We're working on a domain model
  - But we don't yet have enough info for a complete model
- What do we hope to accomplish?
  - Learn as much as possible from the info provided material
  - Reveal questions for later, more detailed follow-up

- Mistakes will be made!

  One of my pet peeves about reading how to do analysis and design in textbooks is that they always make the right decisions at each step.

  It's important to realize that designers do make mistakes and need to back up and reconsider things. (That's why we have the "V" in the ADIV workflow!)

  So I try to be honest and record my analysis examples as a stream-of-consciousness of what I actually went through considering the problem for the first time, including the mistakes.

. . . .

10.1 Problem Statement

Problem Statement

ODU offers a number of courses via the internet. A common requirement among these courses is for a system of online assessment. An assessment is any form of graded question-and-answer activity. Examples include exams, quizzes, exercises, and self-assessments. In preparation for automating such a system, our group has undertaken a study of assessment techniques in traditional classrooms.

An assessment can contain a number of questions. Questions come in many forms, including true/false, single-choice from among multiple alternatives, multiple choices, fill-in-the-blank, and essay. There may be other forms as well.

Students take assessments that are administered by instructors. The students' responses to each question are collected by the instructor, who grades them by comparison to a rubric for each question. The instructor may also elect to provide feedback (written comments), particularly about incorrect responses.

A total score for the assessment is computed by the instructor. If this is a self-assessment, the score is for informational purposes only. For other kinds of assessments, the instructor records the score in his/her grade book.

Information is returned to the student about their performance. At a minimum, the student would learn of their score and any instructor-provided feedback. Depending upon the instructor, students may also receive the questions, a copy of their own responses, and the instructor's correct answer.

. . . .

10.2 Identifying Candidate Classes and Responsibilities

Identifying Candidate Classes and Responsibilities

For the initial list, mark up the description, looking for *noun phrases* and verb phrases .

ODU offers a number of courses via the Internet. A common requirement among these courses is for a system of on-line assessment. An *assessment* is any form of graded question-and-answer activity. Examples include *exams* , *quizzes* , *exercises* , and *self-assessments* . In preparation for automating such a system, our group has undertaken a study of assessment techniques in traditional classrooms.

An assessment can contain a number of *questions* . Questions come in many forms, including *true/false* , *single-choice* from among multiple alternatives, *multiple choices* , *fill-in-the-blank* , and *essay* . There may be other forms as well.

Students take assessments that are administered by *instructors* . The students' *responses* to each question are collected by the *instructor* , who grades them by comparison to a *rubric* for each question. The instructor may also elect to provide *feedback* (written comments), particularly about incorrect responses.

A total *score* for the assessment is computed by the instructor. If this is a self-assessment, the score is for informational purposes only. For other kinds of assessments, the instructor records the score in his/her *grade book* .

Information is returned to the student about their *performance* . At a minimum, the student would learn of their score and any instructor-provided feedback. Depending upon the instructor, students may also receive the questions, a copy of their own responses, and the *instructor's correct answer* .

. . . .

Candidate Classes

assessment, exams, quizzes, exercises, self-assessments, questions, true/false question, single-choice question, multiple choices question, fill-in-the-blank question, essay question, students, instructors, responses, rubric, feedback, score, grade book, information, performance, instructor's answer

. . . .

Candidate Responsibilities

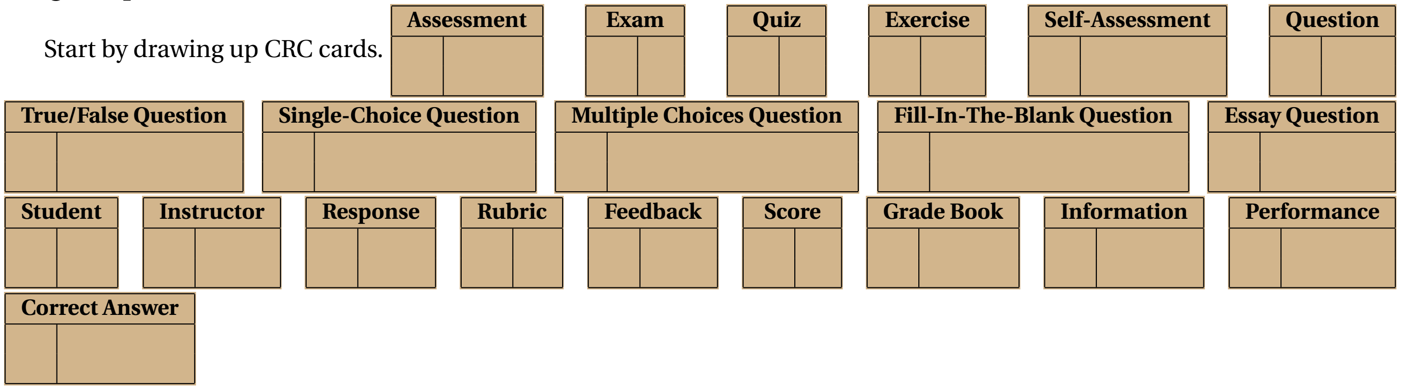
contain (questions), take (assessment), administer, collect (responses), grade, provide (feedback), compute (score), record (score), return (information)

. . . .

10.3 Assign Responsibilities to Classes

Assign Responsibilities to Classes

Start by drawing up CRC cards.



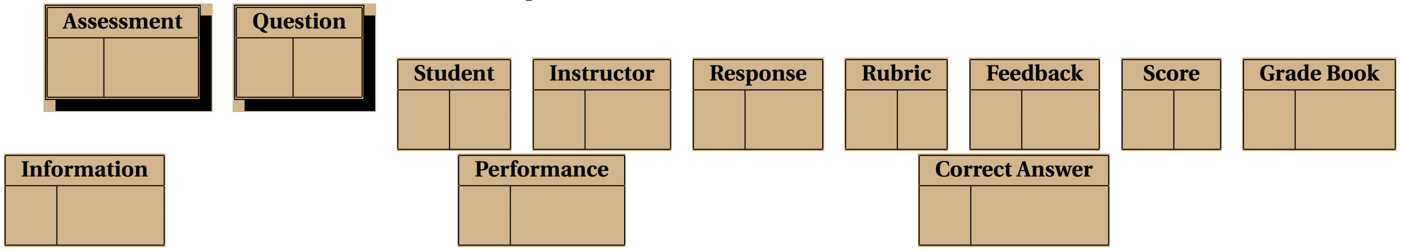
. . . .

10.4 Probable Inheritance Hierarchies

Probable Inheritance Hierarchies

Since all of the various kinds of assessments are likely to have similar responsibilities and collaborators, let's stack their cards for now and treat them as a unit.

We'll do the same with the various kinds of questions.



. . . .

10.5 Fill in the Candidate Responsibilities

Fill in the Candidate Responsibilities

Now fill in the operations known so far:

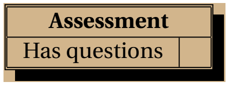
- contain (questions)
- take (assessment),
- administer,
- collect (responses),
- grade,
- provide (feedback),
- compute (score),
- record (score),
- return (information),

. . . .

contain questions

An assessment can contain a number of *questions* .

- This is really a statement about attributes of an assessment



. . . .

taking and administering assessments

Students take assessments that are administered by *instructors* .

- Is this really two separate operations?

  The language (plurals) is a bit tricky.

  – Instructors administer an assessment to an entire class.

  – Each student individually takes the assessment.

Instructor	
administer assessment to group of Students	

Taking or Administering?

- Surprised that I put that in Instructor?

  Remember the basic rule: if A does B to C, then "do B" is usually a responsibility of C

- It would not be a responsibility of the assessment

  – Tests don't administer themselves ITRW

- Could it be a responsibility of the Student?

  – No, the statement says that students "take"assessments,

  – But is "take assessment" simply a synonym for "accept administration of an assessment"?

What's involved in administering an assessment?

The problem statement tells us:

> Students take assessments that are administered by instructors. The students' responses to each question are collected by the instructor, who grades them ... The instructor may also elect to provide feedback (written comments), particularly about incorrect responses.
> A total score for the assessment is computed by the instructor.... Information is returned to the student about their performance.

We're looking at the instructor's *method* for administering an assessment.

- a strong suggestion that administration of an assessment is far more involved than simply having a student "take" it.
  - and that they are, therefore, separate responsibilities.

....

taking an assessment

So we add the student's role into our model:

Student	
take an Assessment	Assessment

Instructor	
administer assessment to group of Students	Student

....

collecting responses

> The students' *responses* to each question are collected by the *instructor*

- This is really just describing the output from the request sent to students asking them to take the assessment.

Student	
take an Assessment: Response	Assessment

....

grading responses

the *instructor* , who grades them by comparison to a *rubric* for each question.

Response	
grade	

Instructor	
administer assessment to group of Students	Student Response

- not a responsibility of the instructor, because we are still tracing out the steps that constitute the instructor's method for administering an assignment

. . . .

Working with Rubrics

We are told there is a separate rubric for each question. So the "comparison" is between a response to a single question and a rubric.

- This highlights the distinction between the response to an assessment and the responses to individual questions.
  - We don't know what the proper terminology here would be, so we use a placeholder and make a note to consult the domain experts.

Response	
has QuestionResponses? grade all question responses via a seq of Rubrics	Rubric

QuestionResponse?	
grade(Rubric): score	

. . . .

Wait a minute...

Response	
has QuestionResponses? grade all question responses via a seq of Rubrics	Rubric

QuestionResponse?	
grade(Rubric): score	

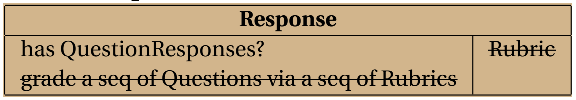
At this point, sanity reasserts itself

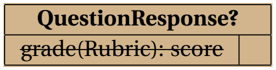
- ITRW, when a student returns an exam sheet or a bluebook (the Response), those things don't grade themselves.

- They're just paper
- And while anthropomorphism is common in OO modeling, that may be going a little too far.

. . . .

What are our options?



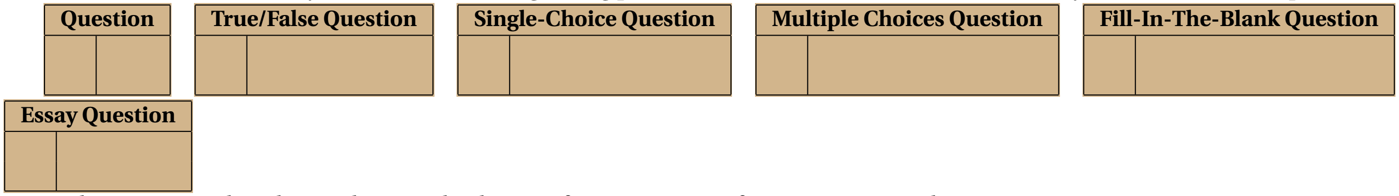


- First thought: the instructor does the grading
  - but it's not really a responsibility of the instructor because no one in this model tells the instructor to take this step
  - it's part of the Instructor's "administer an assessment" method
- But not mentioning it all in the model seems wrong
  - So let's think about what we want to eventually capture

. . . .

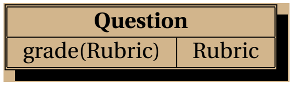
Looking for Variant Behavior

One reason that we really want to model the grading process is that we know that we have many different kinds of questions:



and we suspect that the grading method varies from one type of question to another.

- So perhaps we should say



. . . .

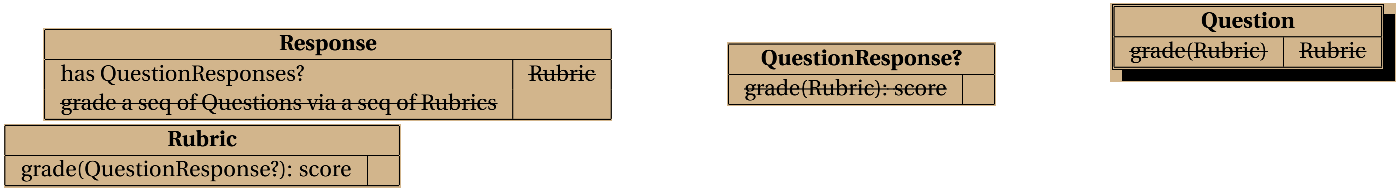
But I Don't Like That Either

. . . and here's why:

- Essay questions can cover all kinds of things.
  - "English" essays where grammar and sentence construction are graded
  - General essays where content is more important than form
  - Mathematical proofs
  - Code an algorithm, and so on
- Are these really the same kind of question?
  - They share lots of behaviors (e.g., they are represented the same way on the printed page, students use the same mechanism for answering them)
  - But the rubrics for grading them are very different
- So maybe it's the rubrics that capture this behavior

. . . .

Grading - revised



- I'm much happier with that
- Can rubrics be "intelligent" or is this unacceptable anthropomorphism again?

  - ITRW, rubrics for essay question, in particular, are often expressed in ways that assume a human intelligence

- We're probably going to wind up with a small deck of Rubric cards for different variants.

. . . .

provide feedback

The instructor may also elect to provide *feedback* (written comments), particularly about incorrect responses.

Graded
Has sco

- Another case of needing to consult the domain experts to find the proper name for a graded question response.

. . . .

computing scores

A total *score* for the assessment is computed by the instructor.

has Gra comput

. . . .

recording grades

the instructor records the score in his/her *grade book*.

record a

Instructor	
administer assessment to group of Students	Student Response Grade Book

. . . .

returning information

Information is returned to the student about their *performance* .

- It's a pretty good bet that we *don't* want a class with as vague a name as "Information".
- But we've already encountered the concept under a better name

  The clue is the description: "At a minimum, the student would learn of their score and any instructor-provided feedback."

Student	
take an Assessment: Response receive a GradedResponse	Assessment

. . . .

10.6 The Story So Far

The Story So Far

Assessment	
Has questions	

GradeBook	
record a score for a Student on an Assessment	

GradedQuestionResponse?	
Has score, feedback	

Question	
~~grade(Rubric)~~	~~Rubric~~

GradedResponse?	
has GradedQuestionResponses? compute total score	

Instructor	
administer assessment to group of Students	Student Response Grade Book

QuestionResponse?	
~~grade(Rubric): score~~	

Response	
has QuestionResponses? ~~grade a seq of Questions via a seq of Rubrics~~	~~Rubric~~

Rubric	
grade(QuestionResponse): score	

Student	
take an Assessment: Response receive a GradedResponse	Assessment

We might (cautiously) question whether some of the empty cards represent classes that we need to retain in the model.

- But we're still very early in the discovery process

. . . .

10.6.1 That's Far Enough for Now

That's Far Enough for Now

Right now, we have as many questions as answers.

- but finding useful questions is part of the process
- We can't go much further without more info
  - . . . and it's very dangerous to start making stuff up based on intuition about how we think the program could work.

. . . .

Chapter 11

UML Class Diagrams

UML Class Relationship Diagrams

Next we turn to more formal documentation that you can expect to show outside your team and probably to retain for the lifetime of the project.

We will employ UML notation for this.

- UML notation has quickly become an industry standard
  - with applications outside of "traditional" software engineering: e.g., DBs

. . . .

UML Diagrams

UML provides a number of different diagrams

- *use case diagrams*
- *class diagrams*

- interaction diagrams
  - *sequence diagrams*
  - *collaboration diagrams*
- package diagrams
- state diagrams
- activity diagrams
- deployment diagrams

We'll look eventually at the ones shown in *italics*, starting with class diagrams.

. . . .

Diagramming in Context

The diagrams are part of an overall documentation strategy

- We don't draw diagrams to stand by themselves.
  - Instead, like the diagrams that appear in, say, your textbook, each one is intended as part of a larger document, much of which will be explanatory text.
  - Furthermore, we draw diagrams for the same reasons we write sentences and paragraphs, to *communicate* some specific idea.
    - If a diagram is too complicated to be understood, it's no more use than a sentence that is too complicated to be understood.
    - If a diagram is too vague to convey any definite meaning, it has no more value than an equally vague sentence.
- The success or failure of a diagram lies entirely in its ability to communicate what the author intended.
- The diagrams are actually the representation of a formal "language" and are expected to make statements that are meaningful according to the rules of that language.

. . . .

11.1 Class Diagrams

Class Diagrams

A class diagram describes the *types* of objects in the system and selected static *relationships* among them. The relationships can be

- generalization (e.g., a Librarian is a specialized kind of Library Staff)
- association (e.g., a Patron may have up to 20 Publications checked out at one time)

. . . .

Perspectives

A diagram can be interpreted from various perspectives:

- *Conceptual*: represents the concepts in the domain
  - at most loosely related to the software classes that will implement them
- *Specification*: focus is on the interfaces of ADTs in the software
- *Implementation*: describes how classes will implement their interfaces

The perspective affects the amount of detail to be supplied and the kinds of relationships worth presenting.

. . . .

The choice of perspective depends on how far along you are in the development process. During the formulation of a domain model, for example, you would seldom move past the conceptual perspective. Analysis models will typically feature a mix of conceptual and specification perspectives. Design model development will typically start with heavy emphasis on the specification perspective, and evolve into the implementation perspective.

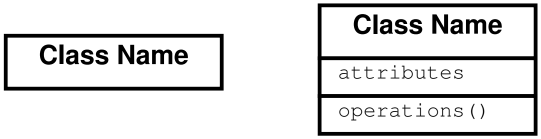
The choice of perspective also depends on the context of the document in which it appears. We don't create diagrams just for the sake of creating diagrams. We create them to communicate with someone. Diagrams are a part of an overall documentation process. If you think about the diagrams that appear, for example, in one of your textbooks, you don't see page after page of diagrams without supporting text. Instead, the diagrams are presented along with explanatory text, as a way of clarifying and emphasizing points being made in the overall narrative flow of the document. Good diagrams can reduce the amount of wordy and often technically stilted text that would otherwise be required.

The same is true when documenting a model. If you are trying to explain a concept in your model, an accompanying diagram should be at the conceptual level. If you are trying to explain the details of how objects interact, the specification perspective may be more appropriate. if you are trying to explain an unexpected feature of how you have implemented or plan to implement a feature, you are probably going to portray things from an implementation perspective. Even late in the development of the design model, you may find it necessary to switch back and forth from one perspective to another in order to be an effective communicator.

11.2 A Class, in Isolation

A Class, in Isolation

A class can be diagrammed as

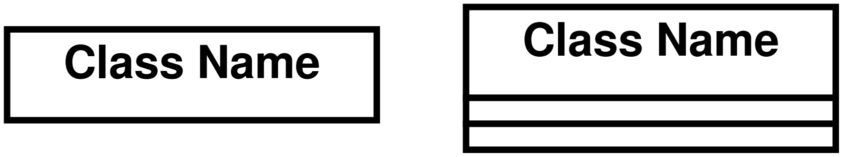


You would use the simpler form in a conceptual perspective, or if the attributes and operations of the class were not relevant to the point of your diagram in specification or implementation perspectives.

. . . .

Even an Empty Section has Meaning

Note that these are *not* equivalent:



- The left diagram says that we aren't discussing the attributes and operations.

- The right diagram says that this class has no relevant attributes and operations.

. . . .

11.2.1 Attributes

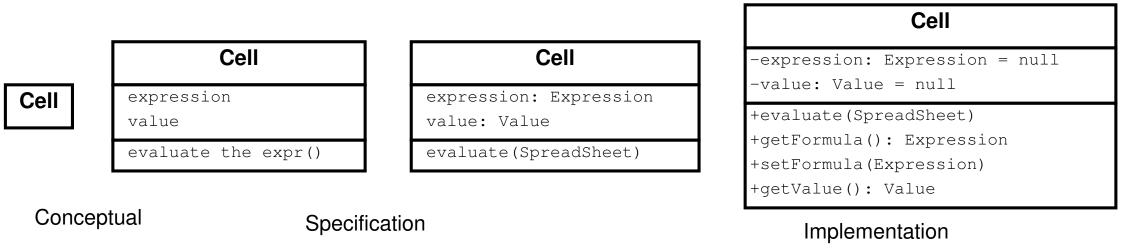
Attributes

Attributes are given as

```
visibility name: type = defaultValue
```

Only the name is required. The others may be added when relevant:

- visibility: + (public), # (protected), or - (private)
  - Only for implementation perspective.
    When you are only talking about concepts or interface specifications, the idea of a "private" anything is irrelevant.



Attributes are usually single values.

- Ones that take multiple values (lists, etc.) are generally represented using associations.

. . . .

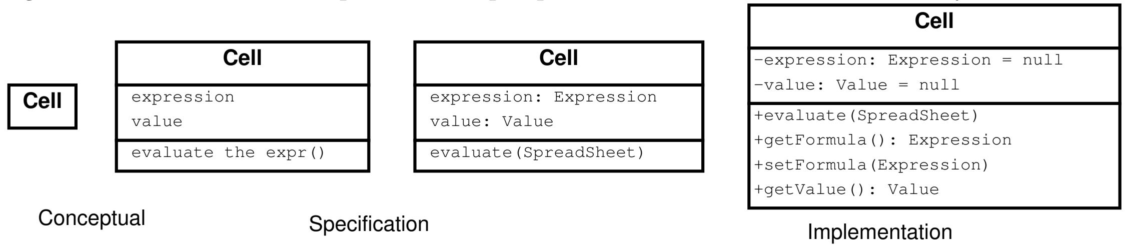
11.2.2 Operations

Operations

Operations are given as

```
visibility name (parameterlist) : return-type
```

Again, the amount of detail depends on the perspective (and on how much has actually been decided)



. . . .

11.3 Generalization Relationships

Generalization Relationships

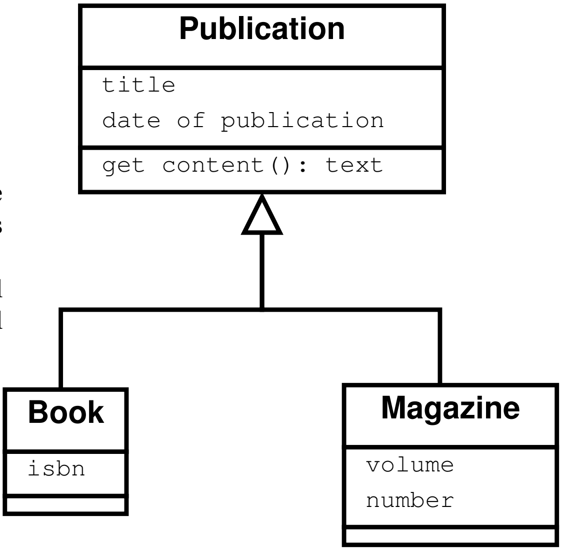
A class (the "*subtype*") is considered to be a *specialized* form of another class (the "supertype") or, alternatively, the supertype is a *generalization* of the subtype if

- conceptual: all instances of the subtype are also instances of the supertype
- specification: the interface of the subtype contains all elements of the interface of the supertype
  - The subtype's interface is said to *conform* to the interface of the supertype
- implementation: the subtype inherits all attributes and operations of the supertype

. . . .

Diagramming Generalization

The UML symbol for this relationship is an arrow with an unfilled, triangular head.



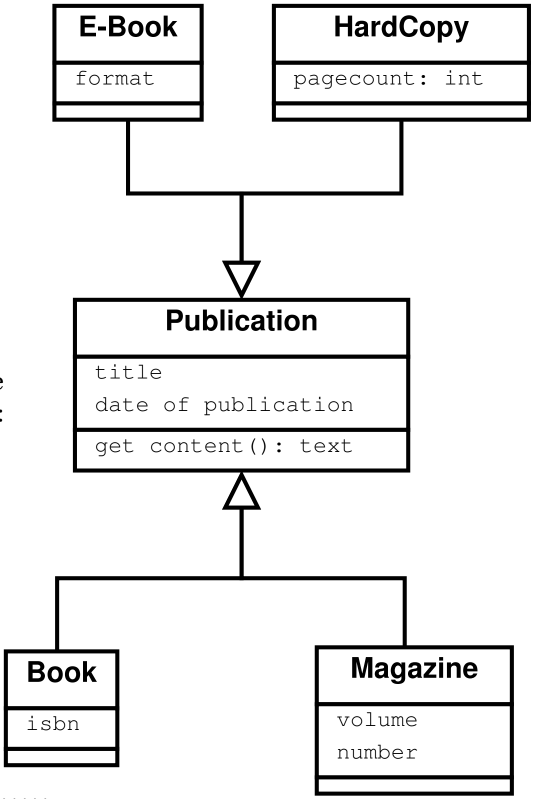
Read this arrow as "is a specialization of", "is a kind of", or "is a". (The latter can be a bit ambiguous however, as we might also say that Webster's Dictionary "is a" Book, but that's not a generalization relationship.

We infer from the relationship that books and magazines have titles and dates of publication and that we can get their content. However, not all publications have ISBNs.

. . . .

Multiple Specializations

Some classes may participate in multiple generalization relationships. The arrows are grouped together to indicate related, mutually exclusive, divisions:



. . . .

11.4 Associations

Associations

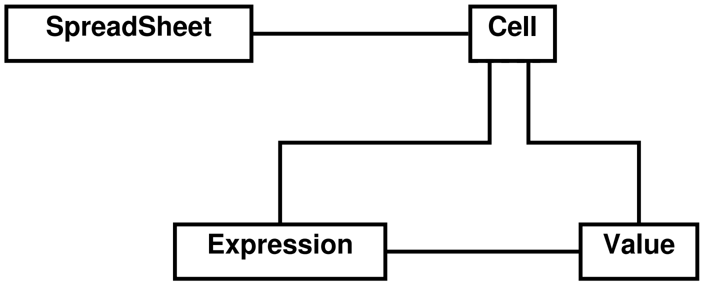
- An *association* is any kind of relationship between instances of classes.

  - Generalization is not an association because it is a relation between the classes themselves, not between their instances (objects).

- In a conceptual perspective, associations represent general relationships. In a specification perspective, associations often denote responsibilities.

. . . .

Diagramming Associations



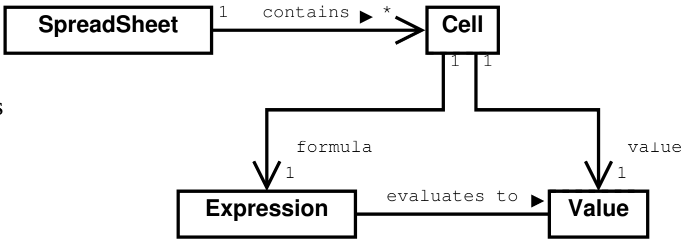
Associations are shown as lines connecting classes.

- Plain associations like these are not particularly useful.

- This one tells us, for example, that some spreadsheets have *some kind* of relationship with *some* cells.
  - Only in very rare circumstances would that be a useful insight into the author's understanding of the spreadsheet world.

. . . .

Decorations

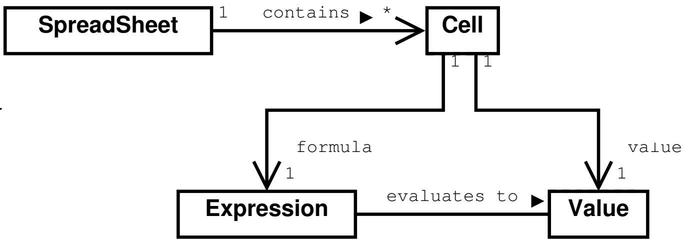


We make the associations meaningful by attaching various decorations.

11.4.1 Naming Your Relationships

Relationship Names

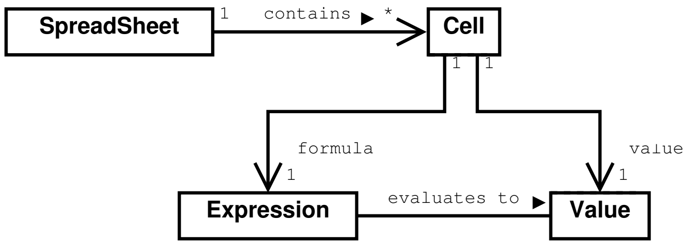
Most important of these are the decorations that name the relationship being shown.



- *Names*: Names of relationships are written in the middle of the association line.

- Good relation names make sense when you read them out loud:
  - "Every spreadsheet contains some number of cells",
  - "an expression evaluates to a value"
- They often have a small arrowhead to show the direction in which direction to read the relationship, e.g., expressions evaluate to values, but values do not evaluate to expressions.

Roles



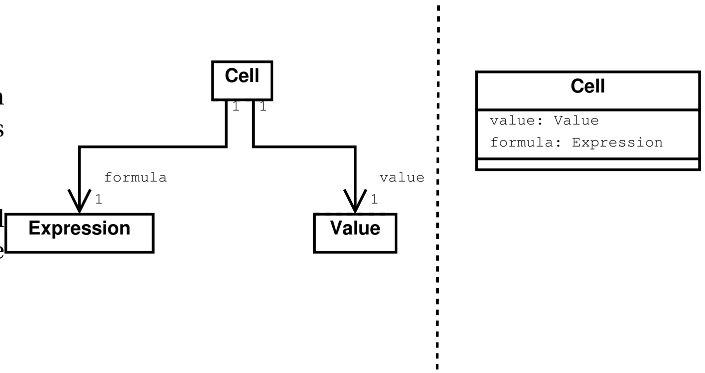
- *Roles*: a role is a directional purpose of an association.

- Roles are written at the ends of an association line and describe the purpose played by that class in the relationship.
  - E.g., A cell is related to an expression. The nature of the relationship is that the expression is the formula of the cell.

Roles and Attributes

In a specification or implementation perspective, roles often correspond to attribute names. These two representations are, in a sense, equivalent.

- If we have conventions for naming attribute retrieval functions (e.g., get/set) , we can infer those from the role name.



Roles versus Attributes

When should you use attributes within a class and when should you use associations with role names?

- Use associations when the relationship is not one-to-one.
- Use associations when you want to target the attribute's class for other associations (e.g., the "evaluates to" association in the earlier diagram.

. . . .

Unnamed Associations are Useless

As a rule, I would argue that *every* association should have either a name or roles

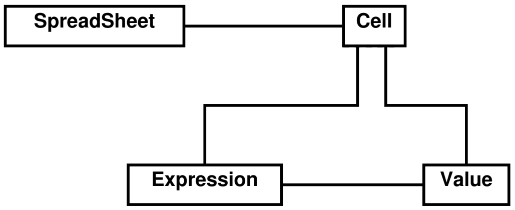
- (or be replaced by one of the *specialized associations* described later).

. . . .

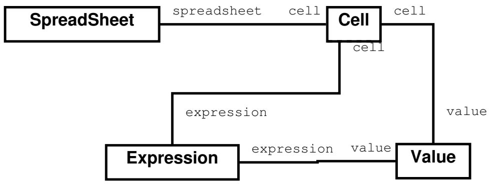
Unnamed Associations are Useless (cont.)

Not everyone agrees.

- Some authors suggest that, if neither is given explicitly, then the association should be considered to have roles named for the classes.
  - In other words, this



would default to



Neither of these is an attractive choice, but, in practice, there are generally better options.

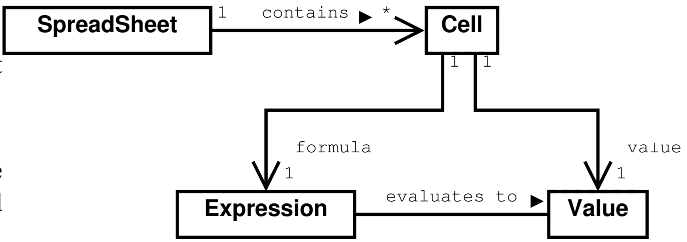
11.4.2 Multiplicity

Multiplicity

Multiplicity indicates how many instances of a class participate in the relationship.
Multiplicity is encoded as:

- `k`: Exactly k instances (where k is an integer or a known constant)
- `k..m`: Some value in the range from k to m (inclusive)
- $*$: Denotes the range 0..infinity. Can also be used on the upper end of a "..", e.g., 1..* means "at least one".

Diagramming Multiplicity

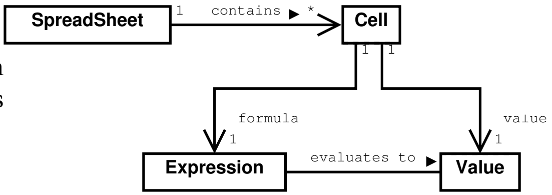


This diagram above indicates that

- each spreadsheet contains any number of cells, but that
- a cell is contained within exactly one spreadsheet.
- Each cell contains exactly one expression and one value, and these values and expressions are not shared with other cells.

11.4.3 Navigability

Navigability



Navigability arrows indicate whether, given one instance participating in a relationship, it is possible to determine the instances of the other class that are related to it.

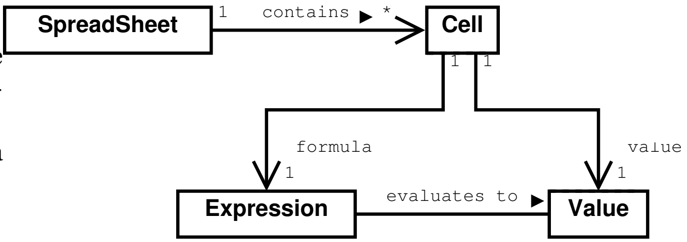
The diagram above suggests that,

- given a spreadsheet, we can locate all of the cells that it contains, but that
  - we cannot determine from a cell in what spreadsheet it is contained.
- Given a cell, we can obtain the related expression and value, but
  - given a value (or expression) we cannot find the cell of which those are attributes.

. . . .

Navigability and Role Names

There's a relationship between navigability and role names.



In perspectives and relationships where the role names are attribute names, we would not have a role name for an un-navigable direction of an association.
Hence, we have no role name naming the cell related to a value or expression.

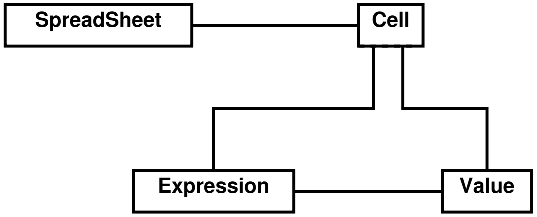
. . . .

Navigability and Perspectives

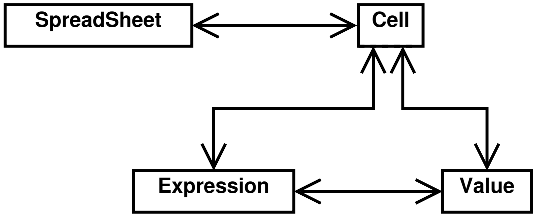
- Navigability is probably not useful in a conceptual diagram.

- In a specification/implementation diagram, if no arrows are shown on an association, navigability defaults to two-way.

  For example, in a specification perspective, this



  defaults to



  - In an implementation perspective, this would imply pointers in each object going to the related objects.

. . . .

11.5 Specialized Associations

Specialized Associations

Certain kinds of associations occur so frequently that they are given special symbols that replace names and (often) role labels.

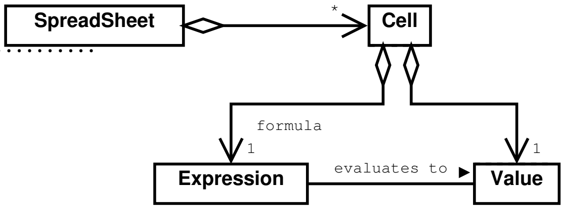
. . . .

11.5.1 Aggregation

Aggregation

Denoted by an arrowhead drawn as an unfilled diamond, *aggregation* can be read as "is part of" or, in the opposite direction as "has a".

This diagram suggests that cells are part of a spreadsheet and that an expression and a value are each part of a cell.



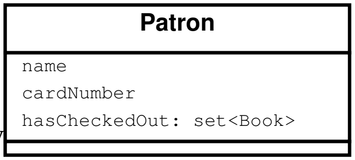
. . . .

Aggregation Details

- The multiplicity at the arrowhead defaults to "1".
- Aggregation is almost never named and roles are only used if the attribute name would be unexpected.
- There is a lot of variation in deciding when to use use aggregation.
  - For example, experts might disagree on whether a Library can be represented as an aggregate of its Librarians and of its Patrons.
  - I recommend reading your suggested aggregations, out loud, as "is part of" and asking yourself whether they make sense.

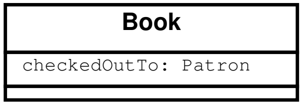
. . . .

Example: Should This Be Aggregation?

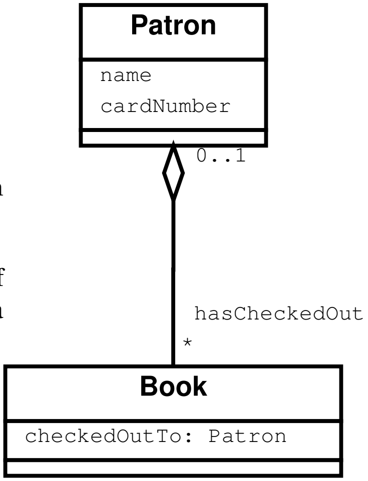


For example, this diagram tries to circumvent our rule that attributes are single-valued by having a "single" set as an attribute.

- That would be OK in an implementation perspective (i.e., in a design model),
- but not in an earlier model/perspective where it prematurely suggests a data structure.

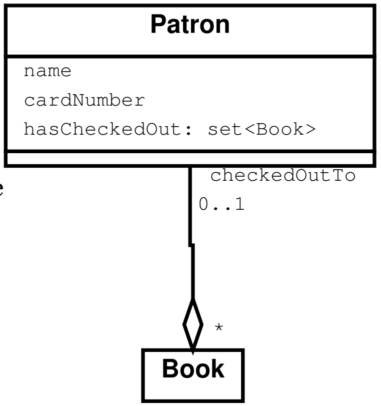


Example: Should This Be Aggregation? (2)



- This alternative would also be OK in a design model.
- But it’s terrible in a conceptual perspective and uncomfortable in a specification perspective.
  - If I read this as "books can be part of a patron", I get a mental picture of books being surgically implanted into people. I would not accept that in a domain or analysis model.

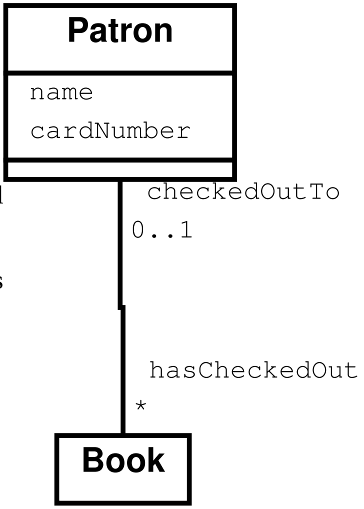
Example: Should This Be Aggregation? (3)



This one is no better in the earlier perspectives.

- If "a patron can be part of a book", does that mean that I will find flat little people pressed between the pages?
  - I'm starting to feel like I will never visit a library again.

Example: Should This Be Aggregation? (4)



So, for conceptual or specification perspectives, I'd avoid aggregation in this case and fall back to a general association.

- (This also captures the bi-directional navigability with a bit more elegance as well.)

....

11.5.2 Composition

Composition

Composition is stronger form of aggregation. It implies that the "lifetime" of the parts is bound to the lifetime of the whole.

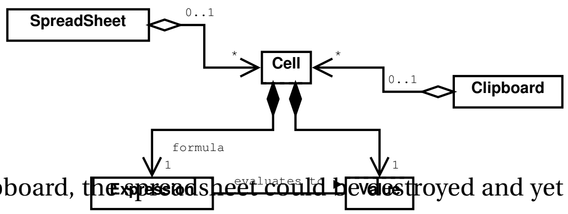
- The usual test to see if composition applies is to ask, "if I delete/destroy the container, do the parts go away as well?"
- Composition is denoted by an arrowhead drawn as a filled-in diamond.

....

Composition Example

This diagram suggests that destroying a cell would also destroy its value and formula. That makes sense if the cell does not share those components with other objects.

However, destroying a spreadsheet does not necessarily destroy its cells.

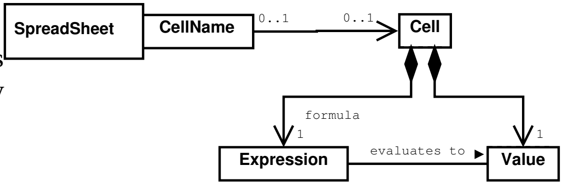


- (We can infer that because, if the cells have been cut or copied to the clipboard, the spreadsheet could be destroyed and yet the cells would remain available for pasting into some other spreadsheet.)

11.5.3 Qualification

Qualification

A qualified association describes a situation in which one class is related to multiple instances of another, but the collection of related instances is "indexed" by a third class.



- This diagram indicates that a spreadsheet provides access to cells retrieved by cell name. It suggests an eventual implementation by something along the lines of a map or table.

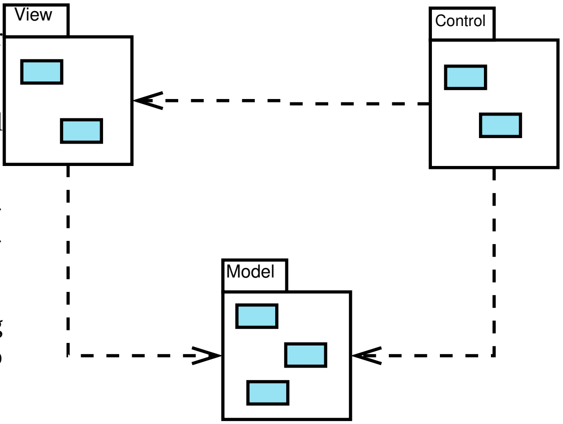
11.5.4 Dependency

Dependency

A class A *depends* on another class B if a change to the interface of B might require alteration of A.

- A dependency is indicated by a dashed line ending at a navigability arrow head.
- Dependencies have little use outside of an implementation perspective.

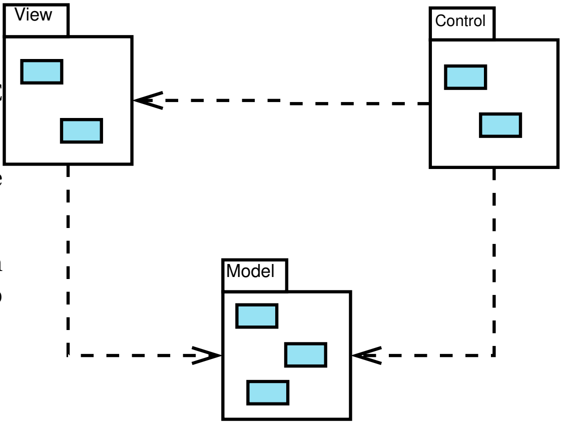
Dependency Example



This diagram shows a system that has been decomposed into three major subsystems.

- The Model is the core data - the stuff that this program is really all about.
- The View is the portion of the code responsible for printing or drawing portrayals of the model data - for example, code to render graphics on a screen as part of a GUI.
- The Controller is the portion of the code responsible for accepting interactive inputs (mouse clicks, key presses, etc) and responding to them.

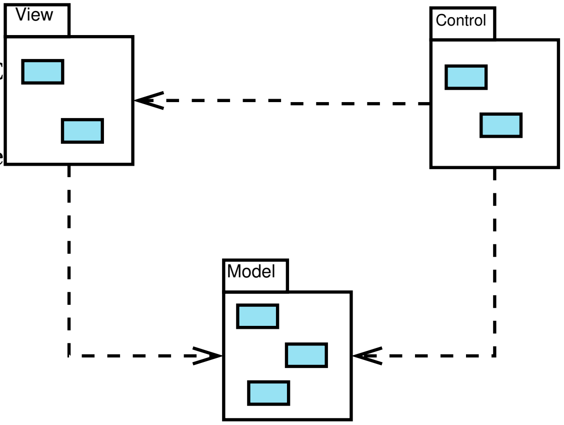
MVC



The dependencies shown here are the defining characteristic of the MVC approach to user interface design:

- the controller depends upon, and therefore can make calls upon, the model
  - (e.g., if someone enters a new data value via a GUI) and upon the view (e.g., someone clicks on a scroll bar or zoom button to alter the view without actually affecting the data).

MVC

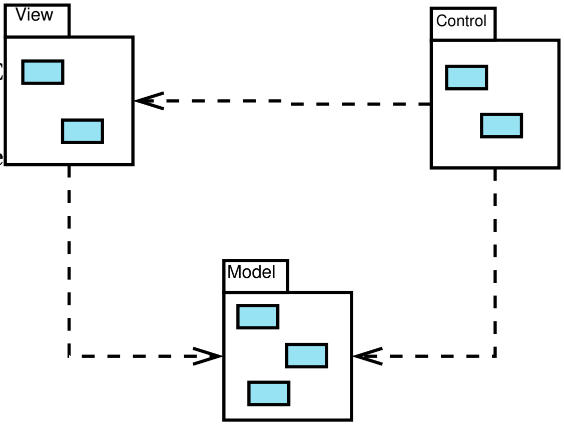


The dependencies shown here are the defining characteristic of the MVC approach to user interface design:

- the controller depends upon, and therefore can make calls upon, the model
- Neither the view nor the model depend on the controller.
- The view depends on the model
  - you can't draw something without being able to look at it.

. . . .

MVC



The dependencies shown here are the defining characteristic of the MVC approach to user interface design:

- the controller depends upon, and therefore can make calls upon, the model
- Neither the view nor the model depend on the controller.
- The view depends on the model
- But the model does not depend on the view.

The advantage of this interface design is that we can radically change the user interface without altering the core computations (the model).

. . . .

11.6 Other Class Diagram Elements

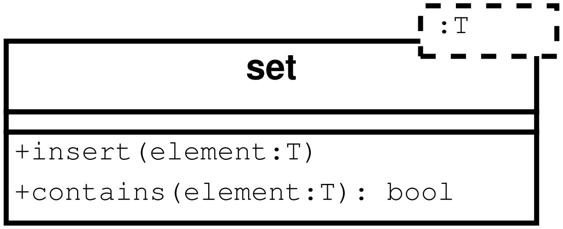
Other Class Diagram Elements

We won't use these as often, but you should be able to recognize them.

. . . .

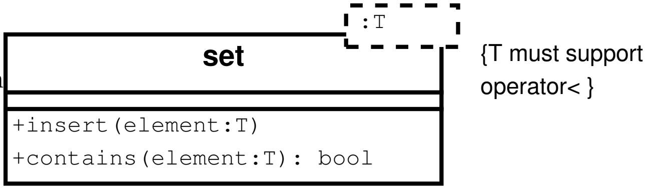
Parameterized Classes

Used to represent templates and similar concepts.



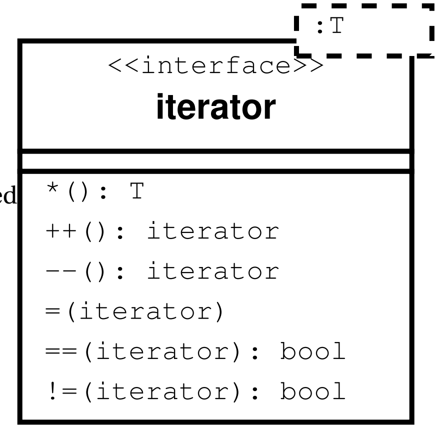
Constraints

Constraints can be added almost any place by writing them within brackets.



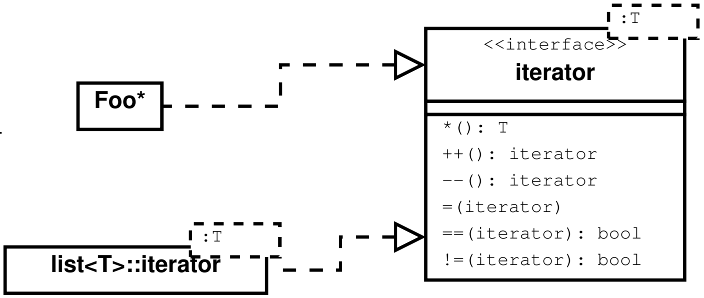
Stereotypes

Stereotypes are an extension mechanism built in to UML. Written within <<...>>, these are labels used to indicate that you are deviating slightly from the standard interpretation of a UML construct.



For example, an *interface* is a collection of related operations but is not a full-fledged class. We represent it as a variation on the normal interpretation of a class.

Satisfies / Realizes



Another specialized association, this is actually a combination of dependency with the arrowhead of generalization.

11.7 Drawing UML Class Diagrams

Drawing UML Class Diagrams

In theory, you can use almost any drawing tool.

- But general-purpose tools offer you more opportunities to mess up
- Better to use a tool that “understands” UML

On our systems, we have

- dia, an open-source product (Windows & Linux)
  - recommended
- Visio, a commercial product from Microsoft (Windows)
  - Impossible to do some standard UML things
    - omit visibility markers
  - Frustrating to do others:
    - getting a simple 1-box class,
    - using user-defined types for attributes & function parameters

. . . .

Data Entry Rather than Direct Formatting

- Both tools work by allowing you to create a class box, then editing its properties
  - These properties include attributes and operations

. . . .

Exporting the Final Product

- As a general rule, you edit and save your documents in the tool’s native file format
- Then export as graphics to be pasted into your documentation

- Use vector graphics formats whenever possible
  - e.g., Use: PDF, EPS, SVG, WMF, EMF
    Best choice depends on what your document tool can import
  - Avoid: GIF, BMP, , PNG
  - Especially avoid "lossy" raster formats: JPEG

. . . .

Chapter 12

Example: Class Diagrams (Domain Model)

Continuing with our Domain Model

First, a quick recap:

. . . .

Problem Statement

ODU offers a number of courses via the internet. A common requirement among these courses is for a system of online assessment. An assessment is any form of graded question-and-answer activity. Examples include exams, quizzes, exercises, and self-assessments. In preparation for automating such a system, our group has undertaken a study of assessment techniques in traditional classrooms.

An assessment can contain a number of questions. Questions come in many forms, including true/false, single-choice from among multiple alternatives, multiple choices, fill-in-the-blank, and essay. There may be other forms as well.

Students take assessments that are administered by instructors. The students' responses to each question are collected by the instructor, who grades them by comparison to a rubric for each question. The instructor may also elect to provide feedback (written comments), particularly about incorrect responses.

A total score for the assessment is computed by the instructor. If this is a self-assessment, the score is for informational purposes only. For other kinds of assessments, the instructor records the score in his/her grade book.

Information is returned to the student about their performance. At a minimum, the student would learn of their score and any instructor-provided feedback. Depending upon the instructor, students may also receive the questions, a copy of their own responses, and the instructor's correct answer.

. . . .

12.1 What Isn't Captured in Our CRC Cards?

What Isn't Captured in Our CRC Cards?

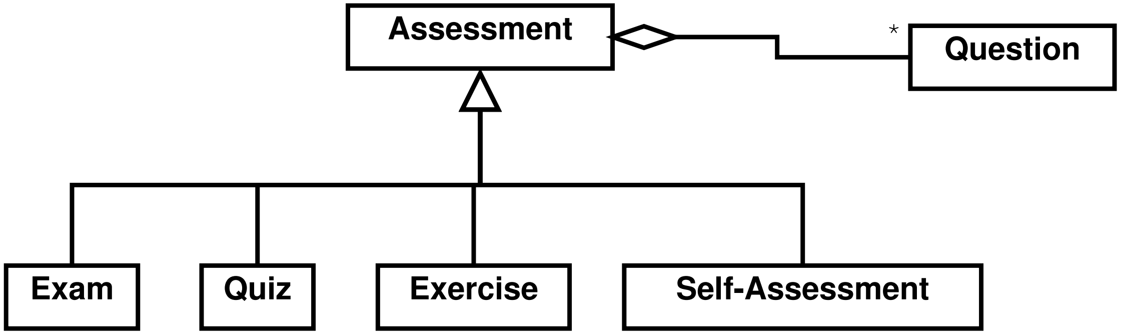
Let's look for things that we learned but that the CRC cards don't really capture.

. . . .

Different Kinds of Assessment

ODU offers a number of courses via the internet. A common requirement among these courses is for a system of online assessment. *An assessment is any form of graded question-and-answer activity. Examples include exams, quizzes, exercises, and self-assessments*. In preparation for automating such a system, our group has undertaken a study of assessment techniques in traditional classrooms.

Suggests a generalization relationship.

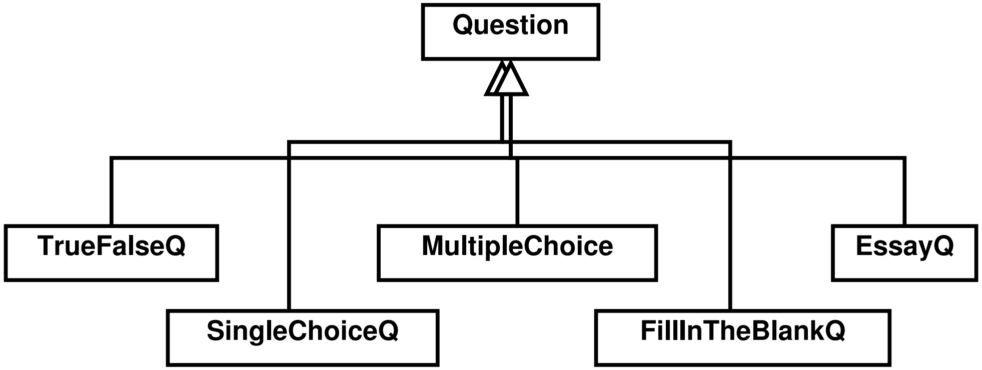


. . . .

Different Kinds of Question

> An assessment can contain a number of questions. *Questions come in many forms, including true/false, single-choice from among multiple alternatives, multiple choices, fill-in-the-blank, and essay.* There may be other forms as well.

Suggests another generalization relationship.

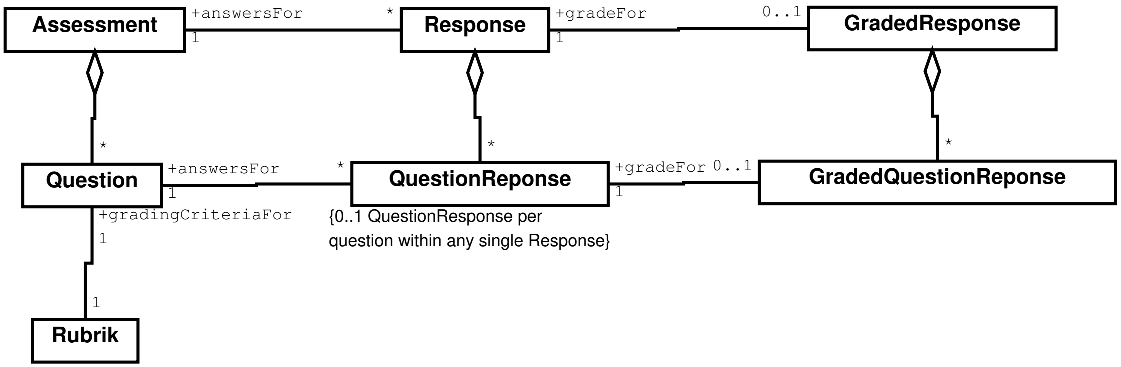


. . . .

Documents Involved in Grading

> Students take assessments that are administered by instructors. The students' responses to each question are collected by the instructor, who grades them by comparison to a rubric for each question. The instructor may also elect to provide feedback (written comments), particularly about incorrect responses.

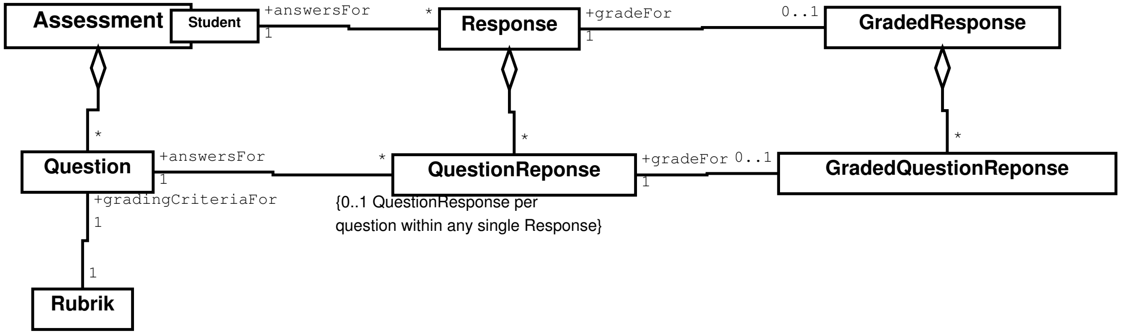
One thing not well captured in our CRC cards is the relation among the documents and the items that make up each document.



. . . .

Documents Involved in Grading (2)

Or, we could even say a little more explicitly how student responses are related to assessments.



. . . .

Part IV

OOP

Chapter 13

Inheritance - the is-a Relationship

13.1 Generalization & Specialization

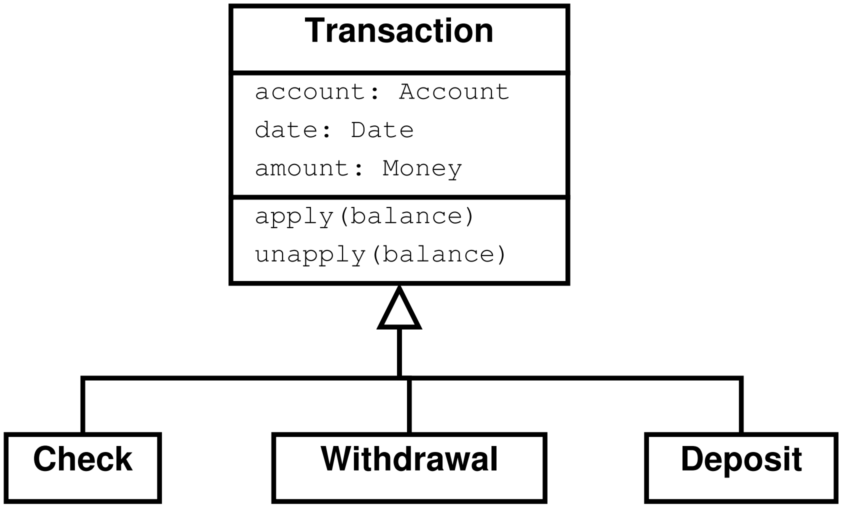
In an earlier lesson, we introduced the idea of a generalization/specialization relationship between classes.

Generalization and Specialization

- Conceptually, class A is a generalization of class B if every B object is also an A object.
  - "everything we say about an" A object "is also true for" a B object.
- At the specification/implementation level, class A is a generalization of class B if B conforms to the public interface of A.

. . . .

Specialization Example



For example, a check and a deposit are actually specializations of the more general concept of a "transaction".

. . . .

If I were to assert, therefore, that every transaction has a date and an amount, we would understand that the same is true for Checks and Deposits.

Or, from the interface point of view, if I were to assert that every `Transaction` has a function member `apply` that takes a `Balance` as its only parameter, then `Checks` and `Deposits` must support the same operation.

13.1.1 Inheritance

In programming languages, generalization is denoted by inheritance and/or subtyping.

Inheritance

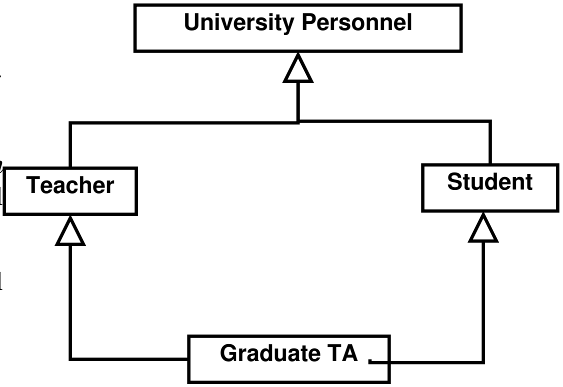
A class C *inherits* from class D if C has all the data members and messages of D.

- C may have additional data members and messages not present in D.

D is called a *base class* of C.

. . . .

Inheritance Example



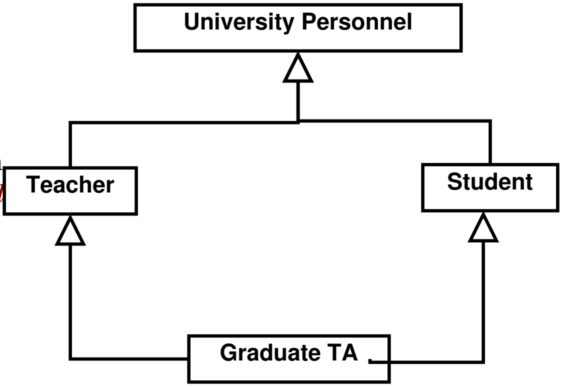
This example suggests that Teachers and Students will inherit from University Personnel.

- So if University Personnel have data members *name* and *uin* (University ID Number), the same will be true of `Teachers` and `Students`.
- However, `Teachers` may have data members not common to all `UniversityPersonnel`, such as a salary.

- `Students` may likewise have data members not common to all `UniversityPersonnel`, such as a grade point average *gpa*.

. . . .

Inheritance Example



- The diagram also suggests that Graduate TAs will inherit from both `Teachers` and `Students`, so `GraduateTAs` will have a *name*, *uin*, *and* a *salary* and a *gpa*.

. . . .

Multiple Inheritance

Inheriting from multiple base classes is called *multiple inheritance*.

It's reasonably common in domain and analysis models, but designers often try to remove it before they get to the stage of coding. That's because multiple inheritance can lead to complications.

- For example, does a `GraduateTA` get one *name* or two?

- One *uin* or two?

. . . .

It's pretty obvious that the TA's name is not changed depending on whether they are doing teacher stuff or student stuff at the time. But it's not hard to imagine that some Universities might use distinct ranges of numbers for teachers than for students, requiring a TA to actually have two different numbers. And, how would you indicate your preference for inheriting one or two copies of a data member in a programming language.

It gets a bit messy, but it can be done in C++. On the other hand, Java disallows multiple inheritance as an unnecessary complication (partly because, as we will see, Java's `interface` construct let's us achieve much of the same flexibility with fewer complications.

13.1.2 Subtyping

Subtyping

A closely related idea:

- A type C is a *subtype* of D if a value of type C may be used in any operation that expects a value of type D.
  - C is called a subclass or subtype of D.

  D is called a superclass or *supertype* of C.

. . . .

What's the Difference?

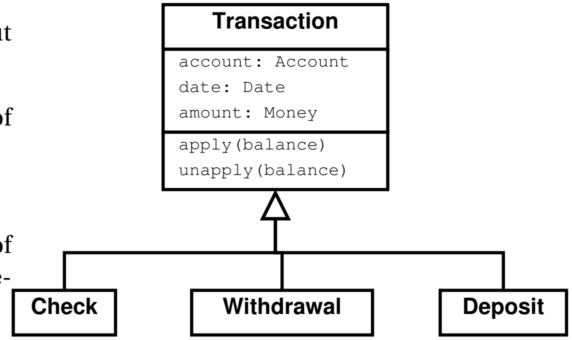
- Inheritance deals with the class's members.
- Subtyping deals with non-members

. . . .

Effects of Subtyping and Inheritance

```
void applyToCurrentBalance (CheckBook cbook, Transaction trans) {
   Balance b = cbook.getCurrentBalance();
   trans.apply (b);
   cbook.setCurrentBalance(b);
}
   ⋮
CheckBook myCheckBook;
Check check;
Transaction transaction;
Balance bal;
```

```
   ⋮
bal = myCheckBook.getCurrentBalance();
transaction.apply (bal);                        ❶
check.apply (bal);                              ❷
applyToCurrentBalance(myCheckBook, transaction); ❸
applyToCurrentBalance(myCheckBook, check);      ❹
```



For example, the last four statements in the code above are all legal, but the reasons vary:

❶ We can make this call because `apply` is a member function of `Transaction` and `transaction` is of type `Transaction`.

Nothing special there.

❷ We can make this call because `apply` is a member function of `Transaction`, `Check` *inherits* from `Transaction`, and `Check` therefore has that same member function.

❸ We can make this call because `applyToCurrentBalance` is a non-member function that takes a `Transaction` as a parameter, and `transaction` is indeed of type `Transaction`.

❹ Finally, we can make this call because `applyToCurrentBalance` is a non-member function that takes a `Transaction` as a parameter, `check` is of type `Check` which is a *subtype* of `Transaction`, and we can use a subtype object in any operation where an object of the supertype is expected. Inheritance does not come into play in this case, because the function we are looking at is not a member function.

. . . .

C++ Combines Inheritance & Subtyping

In most OOPLs, including C++, inheritance and subtyping are combined.

- A base class is always a superclass.

- An inheriting class is always a subclass.
- A superclass is always a base class.
- An subclass is always an inheriting class.

That makes the distinction between inheritance and subtyping moot in C++.
The same does not hold, however, of Java, where only the first two of the four above statements are true.

. . . .

13.2 Inheritance & Subtyping in C++

Inheritance & Subtyping in C++

The construct

```
class C : public Super {
```

indicates that

- Inheritance
  - C inherits from Super
  - Super is a base class of C
- Subtyping
  - C is a subtype of Super
  - Super is a supertype of C

. . . .

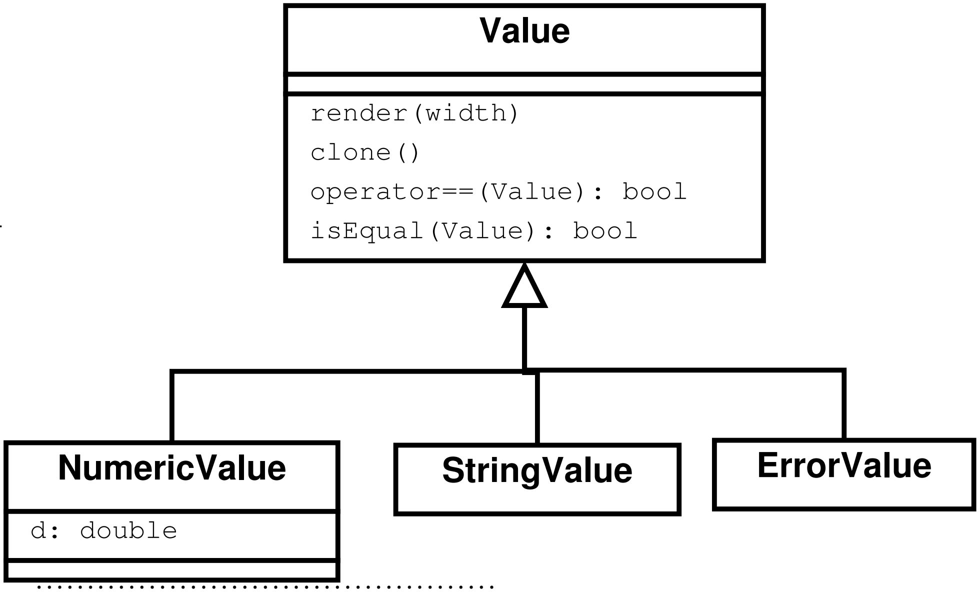
13.2.1 Inheritance Example - Values in a Spreadsheet

Inheritance Example - Values in a Spreadsheet

```
#ifndef CELL_H
#define CELL_H

#include "cellname.h"
#include "observable.h"
#include "observer.h"
#include "strvalue.h"

class Expression;
class Value;
class SpreadSheet;


// A single cell within a Spreadsheet
class Cell: public Observable, Observer
{
public:
  Cell (SpreadSheet& sheet, CellName name);
  Cell(const Cell&);

  ~Cell();

  CellName getName() const;

  const Expression* getFormula() const;
  void putFormula(Expression*);

  const Value* getValue() const;
  const Value* evaluateFormula();

  bool getValueIsCurrent() const;
  void putValueIsCurrent(bool);


  virtual void notify (Observable* changedCell);

private:
  SpreadSheet& theSheet;
  CellName theName;
  Expression* theFormula;
  Value* theValue;
  bool outOfDate;
  static StringValue defaultValue;
};

#endif
```

Every cell in the spreadsheet contains

- a formula (expression)
- a value



Values

The interface for values is

```
#ifndef VALUE_H
#define VALUE_H

#include <string>
#include <typeinfo>

//
// Represents a value that might be obtained for some spreadsheet cell
// when its formula was evaluated.
```

```
//
// Values may come in many forms. At the very least, we can expect that
// our spreadsheet will support numeric and string values, and will
// probably need an "error" or "invalid" value type as well. Later we may
// want to add addiitonal value kinds, such as currency or dates.
//
class Value
{
public:
  virtual ~Value() {}


  virtual std::string render (unsigned maxWidth) const = 0;
  // Produce a string denoting this value such that the
  // string's length() <= maxWidth (assuming maxWidth > 0)
  // If maxWidth==0, then the output string may be arbitrarily long.
  // This function is intended to supply the text for display in the
  // cells of a spreadsheet.


  virtual Value* clone() const = 0;
  // make a copy of this value

protected:
  virtual bool isEqual (const Value& v) const = 0;
  //pre: typeid(*this) == typeid(v)
  //  Returns true iff this value is equal to v, using a comparison
  //  appropriate to the kind of value.

  friend bool operator== (const Value&, const Value&);
};
```

```

inline
bool operator== (const Value& left, const Value& right)
{
  return (typeid(left) == typeid(right))
    && left.isEqual(right);
}

#endif
```

- Values come in many different kinds

. . . .

Numeric Values

Numeric values hold numbers.

```
#ifndef NUMVALUE_H
#define NUMVALUE_H

#include "value.h"

//
// Numeric values in the spreadsheet.
//
class NumericValue: public Value
{
  double d;

public:
  NumericValue():d(0.0)  {}
  NumericValue (double x): d(x) {}
  virtual std::string render (unsigned maxWidth) const;
  // Produce a string denoting this value such that the
  // string's length() <= maxWidth (assuming maxWidth > 0)
  // If maxWidth==0, then the output string may be arbitrarily long.
  // This function is intended to supply the text for display in the
  // cells of a spreadsheet.


  virtual Value* clone() const;

  double getNumericValue() const {return d;}

protected:
  virtual bool isEqual (const Value& v) const;
  //pre: typeid() == v.typeid()
  //  Returns true iff this value is equal to v, using a comparison
  //  appropriate to the kind of value.

};

#endif
```

- We infer by inheritance that `NumericValue` has function members render(), clone(), etc.
- Therefore this code is OK (by inheritance):

```
void foo (NumericValue nv) {
    cout << nv.render(8) << endl;
}
```

- We infer via subtyping that the following code is OK:

```
void foo (Value v; NumericValue nv) {
    if (v == nv) {
       ⋮
```

. . . .

String Values

String values hold numbers.

```
#ifndef STRVALUE_H
#define STRVALUE_H

#include "value.h"

//
// String values in the spreadsheet.
//
class StringValue: public Value
{
  std::string s;
  static const char* theValueKindName;

public:
  StringValue()  {}
  StringValue (std::string x): s(x) {}


  virtual std::string render (unsigned maxWidth) const;
  // Produce a string denoting this value such that the
  // string's length() <= maxWidth (assuming maxWidth > 0)
  // If maxWidth==0, then the output string may be arbitrarily long.
  // This function is intended to supply the text for display in the
  // cells of a spreadsheet.


  std::string getStringValue() const {return s;}

  virtual Value* clone() const;



protected:
  virtual bool isEqual (const Value& v) const;
  //pre: valueKind() == v.valueKind()
  //  Returns true iff this value is equal to v, using a comparison
  //  appropriate to the kind of value.

};

#endif
```

- Note how numeric and string values each add data members that actually store the data they need and that other types of values would find irrelevant.

. . . .

Error Values

Error values store no data at all, but are used as placeholders in a cell whose calculations have failed for some reason.

- (E.g., in any spreadsheet program you have available, try dividing by zero in a cell).

```
#ifndef ERRVALUE_H
#define ERRVALUE_H
```

```
#include "value.h"

//
// Erroneous/invalid values in the spreadsheet.
//
class ErrorValue: public Value
{
  static const char* theValueKindName;

public:
  ErrorValue()  {}

  virtual std::string render (unsigned maxWidth) const;
  // Produce a string denoting this value such that the
  // string's length() <= maxWidth (assuming maxWidth > 0)
  // If maxWidth==0, then the output string may be arbitrarily long.
  // This function is intended to supply the text for display in the
  // cells of a spreadsheet.


  virtual Value* clone() const;

protected:
  virtual bool isEqual (const Value& v) const;
  //pre: valueKind() == v.valueKind()
  //  Returns true iff this value is equal to v, using a comparison
  //  appropriate to the kind of value.

};
```

```
#endif
```

. . . .

13.3 Overriding Functions

Overriding Functions

When a subclass inherits a function member, it may

- inherit the function's body from the superclass, or
- override the function by providing its own body

. . . .

Overriding: Declaring Your Intentions

```
class A {
public:
  void foo();
  void bar();
  void baz();
};

class B: public A {
public:
  void foo();        ❶
  void bar(int k);   ❷
  void bar() const;  ❸
};                   ❹
```

❶ *B* declares that is will override `A::foo()`. *B* inherits the declaration of `foo()` but will provide its own body.

❷ This does not override `A::bar()`.

  – Changing parameters *overloads* a function, but (access to) the original function is unaffected.

❸ This does not override `A::bar()`, either. It also overloads it with different parameter types.

  – The implicit parameter *this* is a `const B*` instead of an `A*`.

❹ Because *B* did not override `A::bar()` or `A::baz()`, it inherits those declarations *and* their bodies from *A*.

. . . .

Access to Functions

- Even when we do override, the original function can be called explicitly:

```
B b;
b.foo();      // calls B::foo()
b.A::foo();   // calls the original A::foo()
```

- This is often useful when the overriding function is supposed to do the same thing as the original *plus* something extra.

```
void B::foo()
{
  A::foo();
  doSomethingExtra();
}
```

. . . .

Example of Overriding

As an example of overriding, consider these four classes, which form a small inheritance hierarchy.

```
class Animal {
public:
  String eats() {return "food";}
  String name() {return "Animal";}
};

class Herbivore: public Animal {
public:
   String eats() {return "plants";}
   String name() {return "Herbivore";}
};

class Ruminants: public Herbivore {
public:
   String name() {return "Ruminant";}
};

class Carnivore: public Animal {
public:
   String name() {return "Carnivore";}
};

void show (String s1, String s2) {
    cout << s1 << " " << s2 << endl;
}
```

Note that several of the inheriting classes override one or both functions in their base class.

Now, suppose we run the following code. What will be printed by each of the show calls?

```
Animal a;
Carnivore c;
Herbivore h;
Ruminant r;
show(a.name(), a.eats());        // AHRC fpgm
show(c.name(), c.eats());        // AHRC fpgm
show(h.name(), h.eats());        // AHRC fpgm
show(r.name(), r.eats());        // AHRC fpgm
```

....

(Try to work this out for yourself before looking ahead.)

Animal Overriding Problem

```
Animal a;
Carnivore c;
Herbivore h;
Ruminant r;
show(a.name(), a.eats());          // Animal food
show(c.name(), c.eats());          // Carnivore food
show(h.name(), h.eats());          // Herbivore plants
show(r.name(), r.eats());          // Ruminant plants
```

Note in particular that, because `Ruminant` does not override its base class's `eats` function, it uses the function body of its base class, `Herbivore`, which itself *has* overridden the `eats()` function of its own base class, `Animal`.

....

Inheritance and Encapsulation

An inheriting class does *not* get access to private data members of its base class:

```
class Counter {
   int count;
public:
   Counter() {count = 0;}
   void increment() {++count;}
   int getC() const {return count;}
};

class HoursCounter: public Counter {
public:
   void increment() {
     counter = (counter + 1) % 24; // Error!
   }
};
```

....

Protected Members

Data members marked as `protected` are accessible to inheriting classes but private to all other classes.

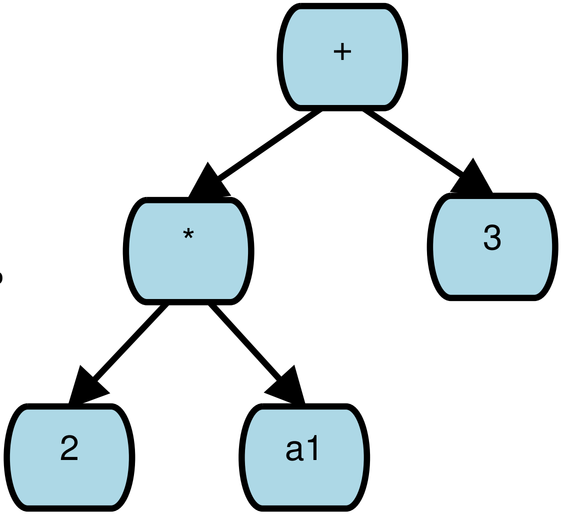
```
class Counter {
protected:
   int count;
public:
   Counter() {count = 0;}
   void increment() {++count;}
   int getC() const {return count;}
};

class HoursCounter: public Counter {
public:
   void increment() {
      counter = (counter + 1) % 24; // OK
   }
};
```

. . . .

13.4 Example: Inheritance and Expressions

Inheritance also plays a part in the spreadsheet in taking care of Expressions

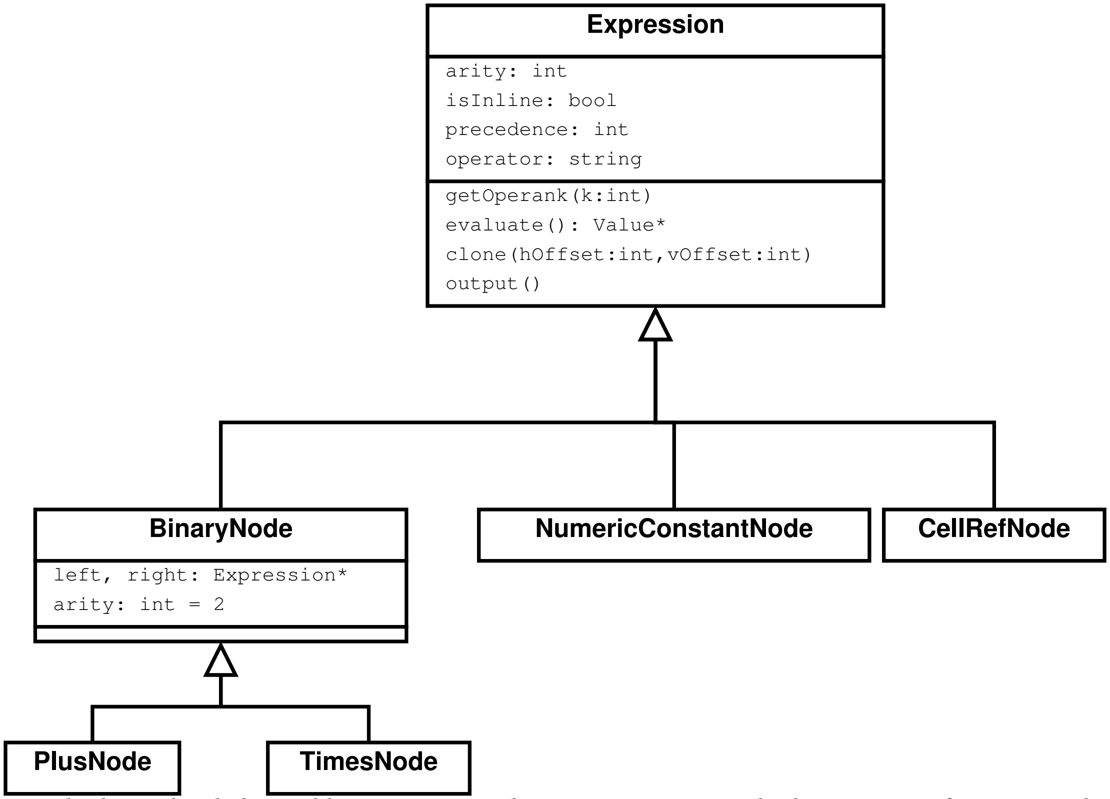
Expression Trees



- An Expression is represented as a tree
  - This tree represents `2*a1 + 3`
- An Expression comes in several parts
  - Internal nodes represent an single operator being applied to some number of operands
    - Each operand is a (sub)Expression
  - Leafs represent constants and variables (cell names)

. . . .

Expression Inheritance Hierarchy



We would expect the lower-level classes like `PlusNode` and `TimesNode` to override the `evaluate` function to do addition, multiplication, etc., or whatever it is that distinctively identifies that particular "kind" of expression from all the other possibilities.

. . . .

Chapter 14

Dynamic Binding

To be considered "object-oriented", a language must support inheritance, subtyping, and dynamic binding. We have seen the first two. Now it's time to look at the third.

If you want to claim that a program is written in an object-oriented style, the code must be designed to take advantage of these three features. Otherwise, it's just a traditionally styled program written in an object-oriented programming language (OOPL).

14.1 Dynamic Binding

Dynamic Binding

Dynamic binding is a key factor in allowing different classes to respond to the same message with different methods.

- Arguably, no language without dynamic binding is an OOPL

. . . .

14.1.1 What is dynamic binding?

What is binding?

Binding is a term used in programming languages to denote the association of information with a symbol. It's a very general term with many applications.

- `a = 2;` is a binding of a value to a variable.
- `String s;` is a binding of a type (`String`) to the variable name (`s`).

. . . .

Binding Function Calls to Bodies

In OOP, we are particularly interested in the binding of a function body (method) to a function call.

Given the following:

```
a = foo(b);
```

When is the decision made as to what code will be executed for this call to `foo`?

. . . .

Compile-Time Binding

In traditionally compiled languages (FORTRAN, PASCAL, C, etc), the decision is made at compile-time.

- Decision is immutable
  - If this statement is inside a loop, the same code will be invoked for `foo` each time.
- Compile-time binding is cheap - there's very little execution-time overhead.

. . . .

Run-Time Binding

In traditionally interpreted languages (LISP, BASIC, etc), the decision is made at run-time.

- Decision is often mutable
  - If this statement is inside a loop, different code may be invoked for foo each time.
- Run-time binding can be expensive (high execution-time overhead) because it suggests that some sort of decision or lookup is done at each call.

. . . .

Dynamic Binding = the Happy Medium?

OOPLs typically feature dynamic binding, an "intermediate" choice in which

- the choice of method is made from a relatively small list of options
  - that list is determined at compile time
  - the final choice made at run-time
- the options that make up that list are organized according to the inheritance hierarchy

. . . .

Dynamic Binding and Programming Languages

- In Java, *all* function calls are resolved by dynamic binding.
- In C++, we can choose between compile-time and dynamic binding. [1]

  Actually, both the class designer and the application programmer have a say in this. Dynamic binding happens only if the class designer says it's OK for a given function, and then only if the application programmer makes calls to that function in a way that permits dynamic binding.

. . . .

[1] This is one of the reasons that C++ is often called a "hybrid" OOPL, as opposed to "pure" OOPLs like SmallTalk.

14.1.2 Dynamic Binding in C++

Dynamic Binding in C++: virtual functions

- A non-inherited function member is subject to dynamic binding if its declaration is preceded by the word `virtual`.
- An inherited function member is subject to dynamic binding if that member in the base class is subject to dynamic binding.
  - Using the word `virtual` in subclasses is optional (but recommended).

. . . .

Dynamic Binding in C++: virtual functions

- Declaring a function as virtual gives programmers permission to call it via dynamic binding.
  - But not *all* calls will be resolved that way.
- Let `foo` be a virtual function member.
  - `x.foo()`, where `x` is an object, is bound at compile time
  - `x.foo()`, where `x` is a reference, is bound at run-time (dynamic).
  - `x->foo()`, where `x` is a pointer, is bound at run-time (dynamic).

. . . .

14.2 An Example of Dynamic Binding

An Animal Inheritance Hierarchy

For this example, we will introduce a simple hierarchy.

```
class Animal {
public:
  virtual String eats() {return "???";}
  String name() {return "Animal";}
};
```

We begin with the base class, `Animal`, which has two functions.

- One of those functions has been marked `virtual`, and so is eligible for dynamic binding.

....

Plant Eaters

Now we introduce a subclass of `Animal` that overrides both those functions.

```
class Herbivore: public Animal {
public:
   virtual String eats() {return "plants";}
   String name() {return "Herbivore";}
};
```

- The "`virtual`" on function `eats` is optional.
  - That function is already virtual because it was declared that way in the base class. But listing the `virtual` in the subclass is a nice reminder to the reader.

....

Cud-Chewers

Now we introduce a subclass of *that* class.

```
class Ruminants: public Herbivore {
public:
   virtual String eats() {return "grass";}
   String name() {return "Ruminant";}
};
```

. . . .

Meat Eaters

And another subclass of the original base class.

```
class Carnivore: public Animal {
public:
   virtual String eats() {return "meat";}
   String name() {return "Carnivore";}
};
```

. . . .

Output Function

It will also be useful in this example to have a simple utility function to print a pair of strings.

```
void show (String s1, String s2) {
         cout << s1 << " " << s2 << endl;
}
```

. . . .

Let's Make Some Calls

Finally, we introduce some application code that makes calls on the member functions of our inheritance hierarchy.

```
Animal a, *paa, *pah, *par;
Herbivore h, *phh;
Ruminant r;
paa = &a; phh = &h; pah = &h; par = &r;

show(a.name(), a.eats());        // AHRC ?pgm
show(paa->name(), paa->eats()); // AHRC ?pgm
show(h.name(), h.eats);          // AHRC ?pgm
show(phh->name(), phh->eats()); // AHRC ?pgm
show(pah->name(), pah->eats()); // AHRC ?pgm
show(par->name(), par->eats()); //AHRC ?pgm
```

. . . .

Note the variety of variables we are using.

- We have three actual objects, `a`, `h`, and `r`, which are of type `Animal`, `Herbivore`, and `Ruminant`, respectively.
- We also have a number of pointers, all of which have names beginning with "`p`".
  - The second letter of each pointer variable name indicates its data type.

    Thus, `pa`, `pah`, and `par` are all of type `Animal*`. `ph` has type `Herbivore*`.
  - These pointer variables are all assigned the address of one of our three actual objects. (The unary prefix operator & in C++ is the "address-of" operator.)

    The third letter in each pointer variable's name indicates what type of object it actually points to. Thus `paa` is of type `Animal*` and actually points to an `Animal` object, `a`.

    `pah`, on the other hand, is of type `Animal*` but actually points to an `Herbivore` object, `h`. (This is possible because of subtyping - we can substitute a subtype object into a context where the supertype object is expected.)

What's the output?

What will be the output of the various `show` calls?

See if you can work this out for yourself before moving on.

. . . .

Output from the Animal Hierarchy

```
Animal a, *paa, *pah, *par;
Herbivore h, *phh;
Ruminant r;
paa = &a; phh = &h; pah = &h; par = &r;

show(a.name(), a.eats());        // Animal ??? ❶
show(paa->name(), paa->eats()); // Animal ??? ❷
show(h.name(), h.eats);          // Herbivore plants ❸
show(phh->name(), phh->eats()); // Herbivore plants ❹
show(pah->name(), pah->eats()); // Animal plants ❺
show(par->name(), par->eats()); //Animal grass ❻
```

. . . .

❶ First we call the functions via the object `a`. Now, `name()` was not declared as a virtual function. So it's not eligible for dynamic binding. This call is resolved by compile-time binding which means that the `Animal::name` body is used and we print "Animal" for the name.

`eats()`, on the other hand, *is* declared as virtual, so the class designer is allowing dynamic binding. However, we are making the call via an object, `a`, not via a pointer or reference, so this call is also resolved at compile-time. The `Animal::eats` body is used and we print "???" for the food.

❷ Next we call the functions via the pointer `paa`. Now, `name()` was not declared as a virtual function. So it's not eligible for dynamic binding. This call is resolved by compile-time binding which means that the compiler looks at the data type of the pointer (`Animal*`) and uses that to choose the function body. The `Animal::name` body is used and we print "Animal" for the name.

`eats()`, on the other hand, *is* declared as virtual, so the class designer is allowing dynamic binding. We are making the call via a pointer, so this call is resolved by dynamic binding. That means that the compiled code, at run-time, follows the pointer out to the heap and uses the data type of the object it actually finds there to determine the choice of function body. In this case, `paa` actually points to an `Animal`, so the `Animal::eats` body is used and we print "???" for the food.

Same output, though the reason is different.

❸ Next we call the functions via the object `h`. Now, `name()` was not declared as a virtual function. So it's not eligible for dynamic binding. This call is resolved by compile-time binding which means that the `Herbivore::name` body is used and we print "Herbivore" for the name.

`eats()`, on the other hand, *is* declared as virtual, so the class designer is allowing dynamic binding. However, we are making the call via an object, `h`, not via a pointer or reference, so this call is also resolved at compile-time. The `Herbivore::eats` body is used and we print "plants" for the food.

❹ Next we call the functions via the pointer `phh`. Now, `name()` was not declared as a virtual function. So it's not eligible for dynamic binding. This call is resolved by compile-time binding which means that the compiler looks at the data type of the pointer (`Herbivore*`) and uses that to choose the function body. The `Herbivore::name` body is used and we print "Herbivore" for the name.

`eats()`, on the other hand, *is* declared as virtual, so the class designer is allowing dynamic binding. We are making the call via a pointer, so this call is resolved by dynamic binding. That means that the compiled code, at run-time, follows the pointer out to the heap and uses the data type of the object it actually finds there to determine the choice of function body. In this case, `phh` actually points to an `Herbivore`, so the `Herbivore::eats` body is used and we print "plants" for the food.

❺ Next we call the functions via the pointer `pah`. Now, `name()` was not declared as a virtual function. So it's not eligible for dynamic binding. This call is resolved by compile-time binding which means that the compiler looks at the data type of the pointer (`Animal*`) and uses that to choose the function body. The `Animal::name` body is used and we print "Animal" for the name.

`eats()`, on the other hand, *is* declared as virtual, so the class designer is allowing dynamic binding. We are making the call via a pointer, so this call is resolved by dynamic binding. That means that the compiled code, at run-time, follows the pointer out to the heap and uses the data type of the object it actually finds there to determine the choice of function body. In this case, `pah` actually points to an `Herbivore`, so the `Herbivore::eats` body is used and we print "plants" for the food.

This, for the first time, shows that dynamic binding can make a different decision than we would have arrived at via compile-time binding.

❻ Finally, we call the functions via the pointer `par`. Now, `name()` was not declared as a virtual function. So it's not eligible for dynamic binding. This call is resolved by compile-time binding which means that the compiler looks at the data type of the pointer (`Animal*`) and uses that to choose the function body. The `Animal::name` body is used and we print "Animal" for the name.

`eats()`, on the other hand, *is* declared as virtual, so the class designer is allowing dynamic binding. We are making the call via a pointer, so this call is resolved by dynamic binding. That means that the compiled code, at run-time, follows the pointer out to the heap and uses the data type of the object it actually finds there to determine the choice of function body. In this case, `par` actually points to a `Ruminant`, so the `Ruminants::eats` body is used and we print "grass" for the food.

When does dynamic binding take us to a different function body that would compile-time binding? Whenever a pointer or references points to a value that is of a subtype of the pointer's declared target type.

14.3 Why is Dynamic Binding Important?

Why is Dynamic Binding Important?

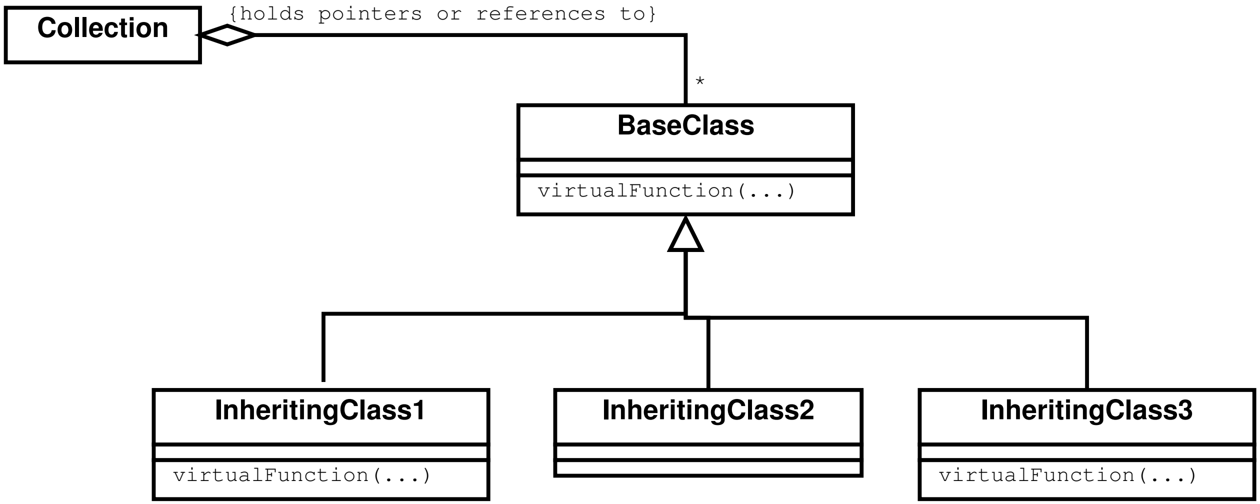
Dynamic binding lets us write application code for the superclass that can be applied to the subclasses, taking advantage of the subclasses' different methods.

....

14.3.1 The Key Pattern to All OOP

Collections of Pointers/References to a Base Class

Suppose we have an inheritance hierarchy:



and that we have a collection of pointers or references to the *BaseClass*.

. . . .

The Key Pattern to All OOP

Then this code:

```
BaseClass* x;
for (each x in collection) {
  x->virtualFunction(...);
}
```

uses dynamic binding to apply subclass-appropriate behavior to each element of a collection.

- Each time around the loop, we extract a pointer from the collection. Thanks to subtyping, that pointer could be pointing to something of type `BaseClass` or to any of its subclasses.

- But when we call `virtualFunction` through that pointer, the runtime system uses the data type of the thing pointed to determine which function body to invoke. If we have enough subclasses, we could wind up doing a different function body each time around the loop.

Study this pattern. Once you understand this, you have grasped the essence of OOP!

. . . .

14.3.2 Examples of the key pattern

Examples of the key pattern

There are lots of variations on this pattern. We can use almost any data structure for the collection.

. . . .

Example: arrays of Animals

```
Animal** animals = new Animal*[numberOfAnimals];
   ⋮
for (int i = 0; i < numberOfAnimals; ++i)
   cout << animals[i]->name() << " "
        << animals[i]->eats() << endl;
```

. . . .

Example: Linked Lists of Animals (C++)

```
struct ListNode {
   Animal* data;
   ListNode* next;
};
ListNode* head; // start of list
   ⋮
for (ListNode* current = head; current != 0; current = current->next)
   cout << current->data->name() << " "
        << current->data->eats() << endl;
```

Example: vector of Animals

```
vector<Animal*> animals;
   ⋮
for (int i = 0; i < animals.size(); ++i)
   cout << animals[i]->name() << " "
        << animals[i]->eats() << endl;
```

Example: Trees of Animals

```
struct TreeNode {
   Animal* data;
   TreeNode* leftChild;
   TreeNode* rightChild;
};
TreeNode* root;

void printTree (const TreeNode* t)
{
  if (t != 0) {
    printTree(t->leftChild);
    cout << t->data->name() << " "
         << t->data->eats() << endl;
    printTree(t->rightChild);
  }
}
   ⋮
printTree(root);
```

This example is a touch more subtle. There's no loop, but the essential idea is the same. We are still iterating over a collection (in this case, using recursive calls), obtaining at each step a pointer that can point to any of several types in an inheritance hierarchy, and using that pointer to invoke a virtual function.

14.4 Examples

14.4.1 Example: Spreadsheet – Rendering Values

Example: Spreadsheet – Rendering Values

Continuing our earlier example:

- Every `Cell` holds a Value.
- Every `Value` can be rendered into a string of a given max width. (See the render function in `value.h`.
- Pairs of `Values` can be compared for equality
- Numeric, String, and Error values are some of the possible `Values`

. . . .

Displaying a Cell

Here is the code to draw a spreadsheet on the screen.

```
void NCursesSpreadSheetView::redraw() const
{
  drawColumnLabels();
  drawRowLabels();

  CellRange shown = showing();
  for (CellName cn = shown.first();
       shown.more(cn); cn = shown.next(cn))
    drawCell(cn);
}
```

After drawing the column and row labels, a call is made to `showing()`. That function returns a rectangular block of cell names (a `CellRange`) representing those cells that are currently visible on the screen, taking into account the window size, where we have scrolled to, etc. We have a loop that goes through the collection of cell names, invoking `drawCell` on each one.

- This is not a virtual call. But if we look *inside* `drawCell`...

. . . .

drawCell

```
void NCursesSpreadSheetView::drawCell
    (CellName name) const
{
  string cellValue;
  Cell* c = sheet.getCell(name);
  const Value* v = c->getValue();
  if (v != 0)
   {
    cellValue = v->render(theColWidth);
   }
  centerStringInWidth (cellValue,
                       theColWidth);
  // . . . show cellValue on screen . . .
}
```

Here we can see that, from the spreadsheet, we get the cell with the given name. Then from that cell we get a pointer to a value. From that pointer we call `render`.

render()

Now `render` in `value.h` is virtual, and various bodies implementing it can be found in classes like

```
std::string NumericValue::render (unsigned maxWidth) const
  // Produce a string denoting this value such that the
  // string’s length() <= maxWidth (assuming maxWidth > 0)
  // If maxWidth==0, then the output string may be arbitrarily long.
  // This function is intended to supply the text for display in the
  // cells of a spreadsheet.
{
  char buffer[256];
  for (char precision = ’6’; precision > ’0’; --precision)
    {
      if (maxWidth > 0)
    {
      sprintf (buffer, "%.1u", maxWidth);
    }
      else
    buffer[0] = 0;
      string format = string("%") + buffer + "." + precision + "g";
      int width = sprintf (buffer, format.c_str(), d);
      if (maxWidth == 0 || width <= maxWidth)
    {
      string result = buffer;
      result.erase(0, result.find_first_not_of(" "));
      return result;
    }
    }
  return string(maxWidth, ’*’);
}
```

, *NumericValue*,

```
std::string StringValue::render (unsigned maxWidth) const
  // Produce a string denoting this value such that the
  // string’s length() <= maxWidth (assuming maxWidth > 0)
  // If maxWidth==0, then the output string may be arbitrarily long.
  // This function is intended to supply the text for display in the
  // cells of a spreadsheet.
{
  if (maxWidth == 0 || maxWidth > s.length())
    return s;
  else
    return s.substr(0, maxWidth);
}
```

, *StringValue*, and

```
std::string ErrorValue::render (unsigned maxWidth) const
  // Produce a string denoting this value such that the
  // string's length() <= maxWidth (assuming maxWidth > 0)
  // If maxWidth==0, then the output string may be arbitrarily long.
  // This function is intended to supply the text for display in the
  // cells of a spreadsheet.
{
  string s = theValueKindName;
  if (maxWidth == 0 || maxWidth > s.length())
    return s;
  else
    return s.substr(0, maxWidth);
}
```

, *ErrorValue*, .

- So that `render` call will be resolved by dynamic binding, sending us to the proper function body depending on just what kind of value is actually stored in the cell.
- Combine that with the loop in the `redraw` function, and we have a loop going through a collection of pointers (the value pointers inside the cells inside the spreadsheet) and using each pointer to invoke a virtual function.

. . . .

14.4.2 Example: Evaluating a Cell

Example: Evaluating a Cell

- After evaluating a formula in a spreadsheet cell

```
const Value* Cell::evaluateFormula()
{
  Value* newValue = (theFormula == 0)
    ? new StringValue()
    : theFormula->evaluate(theSheet);

  if (theValue != 0 && *newValue == *theValue)
    delete newValue;
  else
    {
      delete theValue;
      theValue = newValue;
      notifyObservers();
    }
  return theValue;
}
```

we check to see if the value obtained is equal to *theValue* already stored in that cell.

  – If the values are equal, we simply discard the newly computed value. We don't need it.
  – But if they are not equal, we need to save the new value in place of the old one and trigger the re-evaluation of any cells that mention this one in *their* formulas. (The exact mechanism for how trigger works will be explored later.)

operator==

Look at the implementation of `operator==` in `value.h`

- It calls `isEqual`, which is a virtual function in *Value*.
  - This gives us an opportunity to select subclass-specific behavior for how to compare two values for equality, which you can see in

```
bool NumericValue::isEqual (const Value& v) const
  //pre: valueKind() == v.valueKind()
  //  Returns true iff this value is equal to v, using a comparison
  //  appropriate to the kind of value.
{
  const NumericValue& vv = dynamic_cast<const NumericValue&>(v);
  return d == vv.d;
}
```

,

```
bool StringValue::isEqual (const Value& v) const
  //pre: valueKind() == v.valueKind()
  //  Returns true iff this value is equal to v, using a comparison
  //  appropriate to the kind of value.
{
  const StringValue& vv = dynamic_cast<const StringValue&>(v);
  return s == vv.s;
}
```

, and

```
bool ErrorValue::isEqual (const Value& v) const
  //pre: valueKind() == v.valueKind()
  //  Returns true iff this value is equal to v, using a comparison
  //  appropriate to the kind of value.
```

```
{
  return false;
}
```

.

- Can you see how the key pattern is manifested here?
  - What is the container and what are the contained values?

. . . .

14.5 value.h

```
#ifndef VALUE_H
#define VALUE_H

#include <string>
#include <typeinfo>

//
// Represents a value that might be obtained for some spreadsheet cell
// when its formula was evaluated.
//
// Values may come in many forms. At the very least, we can expect that
// our spreadsheet will support numeric and string values, and will
// probably need an "error" or "invalid" value type as well. Later we may
// want to add addiitonal value kinds, such as currency or dates.
//
class Value
{
public:
  virtual ~Value() {}


  virtual std::string render (unsigned maxWidth) const = 0;
  // Produce a string denoting this value such that the
  // string's length() <= maxWidth (assuming maxWidth > 0)
  // If maxWidth==0, then the output string may be arbitrarily long.
  // This function is intended to supply the text for display in the
  // cells of a spreadsheet.


  virtual Value* clone() const = 0;
  // make a copy of this value

protected:
  virtual bool isEqual (const Value& v) const = 0;
  //pre: typeid(*this) == typeid(v)
  //  Returns true iff this value is equal to v, using a comparison
  //  appropriate to the kind of value.

  friend bool operator== (const Value&, const Value&);
};

inline
bool operator== (const Value& left, const Value& right)
{
  return (typeid(left) == typeid(right))
    && left.isEqual(right);
}

#endif
```

Chapter 15

Making Inheritance Work: C++ Issues

15.1 Base Class Function Members

Base Class Function Members

Even if you override a function, the inherited bodies are still available to you.

```
class Person {
public:
  string name;
  long id;
  void print(ostream& out) {
     out << name << " " << id << endl;}
};

class Student: public Person {
public:
  string school;
  void print(ostream& out) {Person::print(out);
                            out << school << endl;}
}
```

. . . .

Base Class Constructors

This technique is often used in constructors so that subclasses will only need to initialize their own new data members:

```
class Person {
public:
  string name;
  long id;
  Person (string n, long i)
     : name(n), id(i)
  {}
};

class Student: public Person {
public:
  string school;
  Student (string name, long id, School s)
     : Person(name, id), school(s)
  {}
}
```

. . . .

- This is a different use of initialization lists than we have seen before.
  - But is still consistent with the idea that the initialization list is actually a list of constructor calls.

15.2 Assignment and Subtyping

Implementing Data Member Inheritance

Inheritance of data members is achieved by treating all new members as extensions of the base class:

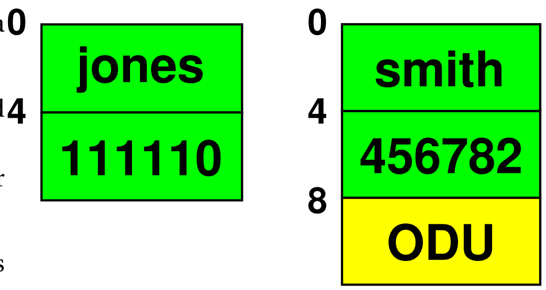
```
class Person {
public:
  string name;
  long id;
};

class Student: public Person {
public:
  string school;
}
```

. . . .

Extending Data Members

- When a compiler processes data member declarations, it assigns a byte offset to each one.
  - In real life, these increase by however manybytes are required to store the previous data member.

    In this example, I'm going to pretend that each data member takes 4 bytes.
- Inherited members occur at the same byte offset as in the base class

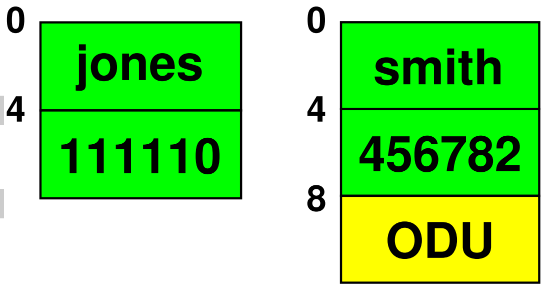


- so code like `p->name` can translate the same whether `p` points to a Person or a Student.
  - `p->name` is translated as "add 0 to the address in *p*"
  - `p->id` is translated as "add 4 to the address in *p*"

- And that works for both Smith and Jones!

. . . .

Assignment & Extension



In most OO languages, we can do

```
superObj = subObj;
```

but not

```
subObj = superObj;
```

- Assigning to a superclass object discards the extra data
  - Presumably, `(Smith, 456782)` is still a valid *Person*
  - Even if it loses the information about Smith being a student
    - So this at least can be said to make sense, even if it's not 100% safe.
- Assigning to a subclass object requires the system to invent data.
- If we assign Jones to a student object, what value should the system copy into the `school`?

. . . .

15.3 Virtual Destructors

Virtual Destructors

As we've seen, subclasses can add new data members.

What happens if we add a pointer:

```
class Person {
 public:
   string name;
   long id;
 };

 class Student: public Person {
 public:
   string school;
 }


class GraduateStudent: public Student {
private:
  Transcript* undergradRecords;
public:
   ...
   GraduateStudent (const GraduateStudent& g);
   GraduateStudent& operator= (const GradudateStudent&);
   ~GraduateStudent();
};

GraduateStudent::GraduateStudent (const GraduateStudent& g)
   : name(g.name), id(g.id), school(g.school),
     undergradRecords(new Transcript(*(g.undergradRecords))
{}

GraduateStudent& operator= (const GradudateStudent& g)
{
  if (this != &g)
    {
     Student::operator=(g);
     delete undergradRecords;
     undergradRecords = new Transcript(*(g.undergradRecords));
    }
   return *this;
}

GraduateStudent::~GraduateStudent()
{
  delete undergradRecords;
}
```

and we don't want to share?

. . . .

Deleting Pointers and Inheritance

Consider the following two delete statements:

```
Person* g1 = new GraduuateStudent(...);
GraduateStudent* g2 = new GraduateStudent(...);
   ⋮
delete g1;  // compiler-generated ~Person() is called
delete g2;  // ~GraduateStudent() is called
```

- Both calls are resolved by compile-time binding
  - Therefore the first delete leaks memory - undergraduateRecords is not cleaned up
- Fix would seem to be to force dynamic binding on the destructors

. . . .

Making the Destructor Virtual

The trick is that this has to be done at the top of the inheritance hierarchy

```
class Person {
 public:
   virtual ~Person() {}
   string name;
   long id;
 };

 class Student: public Person {
 public:
   string school;
 }


class GraduateStudent: public Student {
private:
   Transcript* undergradRecords;
public:
    ⋮
   GraduateStudent (const GraduateStudent& g);
   GraduateStudent& operator= (const GradudateStudent&);
   ~GraduateStudent();
};
```

even though,

- at the time we wrote that class, there may have been no obvious need for a destructor
- this seems to violate the Rule of the Big 3
    - We'll look at the other two in just a moment

. . . .

So you have to think ahead - if there's any chance of a non-shared pointer being added in a future subclass, make your destructor virtual.

15.4 Virtual Assignment

Virtual Assignment

If subclasses can introduce new data members, should assignment be virtual so that we can guarantee proper copying of those extended data members?

. . . .

Virtual Assignment Example

```
void foo(Person& p1, const Person& p2)
{
   p1 = p2;
}

GraduateStudent g1, g2;
  ⋮
foo(g1, g2);
```

- If *p1* and *p2* "really" have *underGraduateRecord* fields, shouldn't we make sure those get copied properly during assignment?
  - Seems reasonable in this case.
  - But it means that assignment and copying will behave very differently, which is likely to catch programmers by surprise.

. . . .

What's the Problem with Virtual Assignment?

If you try it, the inherited members aren't what you might expect:

```
class Person {
 public:
   virtual ~Person() {}
   virtual Person& operator= (const Person& p);
   string name;
   long id;
 };

 class Student: public Person {
 public:
   string school;
   virtual Person& operator= (const Person& p); // inherited from Person
   // Student& operator= (const Student& s); // generated by compiler
 }


class GraduateStudent: public Student {
private:
   Transcript* undergradRecords;
public:
   ...
   GraduateStudent (const GraduateStudent& g);
   virtual Person& operator= (const Person& p); // inherited from Person
   // Student& operator= (const Student& s); // inherited from Student
   GraduateStudent& operator= (const GradudateStudent&);
   ~GraduateStudent();
};
```

You actually wind up with multiple overloaded assignment operators in the subclasses.

What's the Problem with Virtual Assignment? (cont.)

- To make this work, you will need to implement both the virtual and the normal operators
- Implementing the virtual one is tricky because you might not get a GraduateStudent on the right:

```
void foo(Student& s1, const Student& s2)
{
  s1 = s2;
}

Student s;
GraduateStudent g;
   ...
foo(g, s); // problem: s has no undergraduateRecords field
```

Recommendation

- There's no clear consensus in the C++ community about making assignment virtual.
- I recommend against it just because it's potentially confusing.
  - Try to avoid using assignment in situations where the "true" data type on the left is uncertain.

15.5 Virtual constructors

Virtual constructors

- Constructors can never be made virtual
- This can lead to problems when we need to create a copy.

. . . .

Example: evaluating a cell reference

```
class CellReferenceNode: public Expression
{ // represents a reference to a cell
private:
  CellName value;

public:
  CellReferenceNode () {}
    <:smvdots:>

// Evaluate this expression
  virtual Value* evaluate(const SpreadSheet&) const;
    <:smvdots:>
```

```
// Evaluate this expression

Value* CellReferenceNode::evaluate(const SpreadSheet& s) const
{
  Cell* cell = s.getCell(value);
  Value* v = (Value*) cell->getValue();
  if (v == 0)
```

```
    return new ErrorValue();
  else
    return v;
}
```

- We would be better off returning a copy of the spreadsheet cell's value rather than the actual one.
  - Each Cell owns (does not share) its Value
  - Cell may therefore delete that Value
    * don't want to risk some other code doing so
- But how do we make a copy?

. . . .

Not like this!

```
Value* theCopy = new Value(*v);
```

- How big is a Value?
- Would lose all data members in *v* required for its particular subtype of *Value*

. . . .

Better, but Not the "OO Way"

```
Value* newCopy;
if (typeid(*v) == typeid(NumericValue)) {
  newCopy = new NumericValue (v->getNumericValue());
} else if (typeid(*v) == typeid(StringValue)) {
  newCopy = new StringValue (v->render(0));
} else if (typeid(*v) == typeid(ErrorValue)) {
  newCopy = new ErrorValue();
  ⋮
```

(We'll see how `typeid` works shortly.)

....

15.5.1 Cloning

Cloning

Solution is to use a simulated "virtual constructor", generally referred to as a `clone()` or `copy()` function.

```
Value* CellReferenceNode::evaluate(const SpreadSheet& s) const
{
  Cell* cell = s.getCell(value);
  Value* v = (Value*)cell->getValue();
  if (v == 0)
    return new ErrorValue();
  else
    return v->clone();
}
```

....

clone()

`clone()` must be supported by all values:

```
class Value {
public:
     ⋮
  virtual Value* clone() const;
     ⋮
};
```

....

Implementing clone()

Each subclass of Value implements `clone()` as a copy construction passed to `new`.

```
Value* NumericValue::clone() const
{
  return new NumericValue(*this);
}


Value* StringValue::clone() const
{
  return new StringValue(*this);
}
```

. . . .

15.6 Downcasting

Suppose that I want to be able to test any two Values to see if they are equal

- We'll define "equal" here as meaning that they are the same kind of value and would appear the same when rendered.

. . . .

Example: Value::==

We want to explicitly require all subclasses of Value to provide this test:

```
class Value {
      ⋮
   virtual bool operator== (const Value&) const;
};
```

The operator == compares two shapes. Its signature is: `(const Value*, const Value&)` ⇒ `bool`

. . . .

Inheriting ==

```
class NumericValue: public Value {
   ⋮
class StringValue: public Value {
    ⋮
```

Both classes inherit the == operator. The signatures are

```
(const NumericValue*, const Value&) ⇒ bool
(const StringValue*, const Value&) ⇒ bool
```

. . . .

Using the Inherited ==

```
NumericValue n1, n2;
StringValue s1, s2;
bool b = (n1 == n2)
      && (s1 == s2)
      && (n1 == s1);
```

The last clause suggests a problem.

- How do we compare values of different subtypes?
- Should we even allow it?

. . . .

Implementing an asymmetric operator

We might implement == for `NumericValue` as:

```
bool NumericValue::operator==
   (const Value& v)
{
  return d == v.d;
};
```

- But in a call like `(n1 == s1)`, `v.d` does not make sense.
  - In fact, this will get a compile error

. . . .

Implementing an asymmetric operator (cont.)

The problem is that we can easily define

```
bool NumericValue::operator== (const NumericValue& v)
```

but

```
bool NumericValue::operator== (const Value& v)
```

seems impossible, as we cannot anticipate all the values that will ever be defined.

. . . .

15.6.1 RTTI

Working around the == asymmetry

The C++ standard defines a mechanism for *RTTI* (Run Time Type Information).

```
bool NumericValue::operator== (const Value& v)
{
  if (typeid(v) == typeid(NumericValue)) {
       const NumericValue &nv =
                   (const NumericValue&)v;
      return d == nv.d;
  } else
    return false;
};
```

- Note that `typeid()` can be applied both to objects and to types.

- But it can only be used with types/objects that have at least one virtual function.

. . . .

RTTI: typeid and downcasting

RTTI also allows you to test to see if `v` is from a subclass of `NumericValue`

```
if (typeid(NumericValue).before(typeid(v))
```

or to perform safe *downcasting*:

```
NumericValue* np = dynamic_cast<NumericValue*>(&v);
if (np != 0)
   {// v really was a NumericValue or
    //    subclass of NumericValue
           ⋮
   }
```

- The term "downcasting" refers to the fact that we are moving "down" in hour inheritance hierarchy (assuming we draw the base class at the tops).
  - Upcasting is always safe (and usually is done implicitly)
  - Downcasting can be dangerous if we don't check to see if the object really is waht we think it will be.

. . . .

Downcasting Should Not Be a Crutch

Downcasting is often a tempting way to patch a poor initial choice of virtual "protocol" functions.

- 95% of the time, it's a bad idea
  - often leads to subtle, hard to trace bugs

Oddly, though, downcasting is far more widely accepted in Java than in C++.

. . . .

15.7 Single Dispatching & VTables

Equality Again

Earlier, we looked at the problem of comparing two spreadsheet `Values`:

```
class Value {
    ⋮
  virtual bool isEqual (const Value&) const;
};
```

```
class NumericValue: public Value {
    ⋮
  virtual bool isEqual (const Value&) const;
};
```

We saw that problems are caused by `NumericValue::isEqual` getting a parameter of type `Value&` rather than `NumericValue&`.

. . . .

Why is this so hard?

Why can't we select the best fit from among:

```
class NumericValue: public Value {
    ⋮
  virtual bool isEqual (const NumericValue&) const;
  virtual bool isEqual (const StringValue&) const;
  virtual bool isEqual (const ErrorValue&) const;
};
```

The answer stems from how dynamic binding is implemented.

. . . .

15.7.1 Single Dispatching

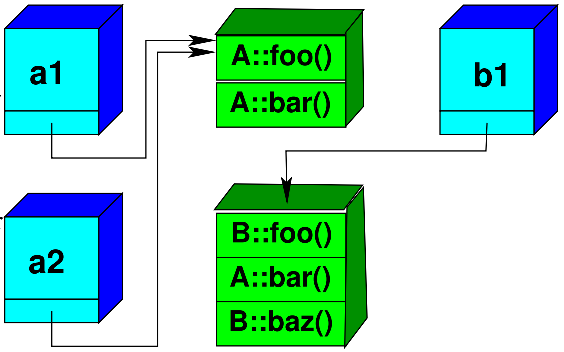
Single Dispatching

Almost all OO languages offer a *single dispatch* model of message passing:

- the dynamic binding is resolved according to the type of the single object to which the message was sent ("dispatched").
  - In C++, this is the object on the left in a call: `obj.foo(...)`
- There are times when this is inappropriate.
  - But it leads to a fast, simple implementation

. . . .

VTables



- Each object with 1 or more virtual functions has a hidden data member.
  - a pointer to a *VTable* for it's class
  - this member is always at a predictable location (e.g., start or end of the object)

. . . .

Compiling Virtual Function Declarations

- Each virtual function in a class is assigned a unique, consecutive index number.
- `(*VTable)[i]` is the address of the class's method for the i'th virtual function.

. . . .

Example of VTable Use

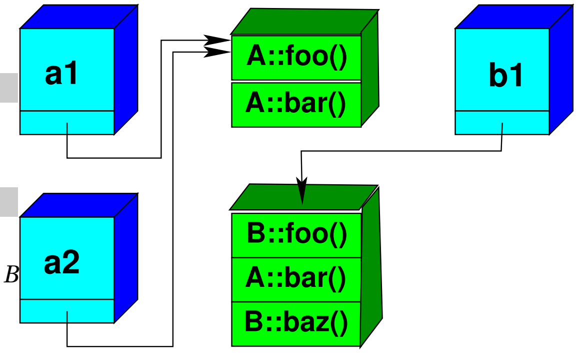
```
class A {
public:
  A();
  virtual void foo();
  virtual void bar();
};

class B: public A {
public:
  B();
  virtual void foo();
  virtual void baz();
};
```

```
A* a = ???;   // might point to an A or a B object
a->foo();
```

`foo()`, `bar()`, and `baz()` are assigned indices 0, 1, and 2, respectively.

. . . .

Example: VTable Structure



- The call `a->foo()` is translated as

```
*(a->VTABLE[0])();
```

- The call `a->bar()` is translated as

```
*(a->VTABLE[1])();
```

Notice that this works regardless of whether *a* points to an *A* object or a *B* object.

- “works” in this case means “does dynamic binding”
- Note that the call `a->baz()` would not compile, so we should not have to worry about going past the end of the shorter vtable.

. . . .

Implementing RTTI

- The address of the VTable is a unique identifier for each class known to the compiler
- This makes the vtable an ideal for implementing RTTI
  – and explains why RTTI is only available for classes with at least one virtual function

. . . .

Chapter 16

Idioms and Patterns

Inheritance, subtyping, and dynamic binding are powerful tools. And, like any new tool, it's not always obvious how and when to use them. The notion that inheritance is the programming language realization of a generalization/specialization relationship helps. It also may help to think of it as the "is a" relationship. When we see these kinds of relationships existing among our classes (in analysis), we should at least consider the possibility that we should be using inheritance in our programs.

But not *every* situation where we see a generalization or as-a relationship in analysis deserves to be programmed as inheritance. There's an old saying, "When the only tool you own is a hammer, every problem begins to resemble a nail." We need to avoid trying to distort our designs into an inheritance mold if it does not actually fit.

Fortunately, you are not the first programmer to be faced with the problem of when to use inheritance. There are selected idioms that have evolved in practice that are worth knowing, both as examples of good ways to use inheritance and as counter-examples of ways to abuse inheritance.

16.1 Inheritance Captures Variant Behavior

Inheritance Captures Variant Behavior

Why bother with inheritance & dynamic binding at all? Because it offers a convenient mechanism for capturing *variant* *behaviors* among different groups of objects. For example, consider the idea of values stored in a spreadsheet. A Spreadsheet can contain many different kinds of values. Some contain numbers, other strings, others might contain dates, monetary amounts, or other types of data. Typically there is a special kind of "error value" that is stored in a cell when we have formulas that do something “illegal”, such as dividing a number by zero or adding a number to a string.

```
string render(int width)
Value* clone()
```

There are some things we expect to be able to do to any value. For values to be useful in a spreadsheet, we must be able to *render* them in a cell whose current column is of some finite width. The render function, in general, could be quite elaborate. For example, given a lot of room, we might be able to render a number as 12509998.0. If we sharing the column containing that cell, though, we might need to render it as 12509998. Shrink it some more and we might need to render it as 1.25E06. Shrink a little more, and we might need to say 1.3E06, then 1E06. Shrink the available space small enough, and a spreadsheet will typically render a number as “***”.

There may be other operations required of all values as well, such as `clone()`.

Now, the information we have provided so far is general to all values. But to actually store a numeric value, we need a data member to hold a number (and a function via which we can retrieve it, though we’ll ignore that for the moment). Similarly, we can expect that, to store a string value, we would need a data member to store the string.

We will assume, therefore, that

- Numeric values have an attribute d of type `double`
- String values have an attribute s of type `string`

. . . .

A pre-OO Approach to Variant behavior

```
class Value {
public:
  enum ValueKind {Numeric, String, Error};
  Value (double dv);
  Value (string sv);
  Value ();
  ValueKind kind;
  double d;
  string s;

  string render(int width) const;
}
```

Now, if we had never heard of inheritance (of if we were programming C, FORTRAN, Pascal, or any of the other pre-OO languages, we might have come up with an ADT something like this.\footnote{Actually, there are ways to make this more memory-efficient in those languages using constructs called “unions” or “variant records”, but they would not change the point of this example. }

- Any given value will presumably have something useful stored in `d` *or* in `s`, but not in both.
    - In the case of an error value, it may not have anything useful in either one.

. . . .

A pre-OO Approach to Variant behavior

```
class Value {
public:
  enum ValueKind {Numeric, String, Error};
  Value (double dv);
  Value (string sv);
  Value ();

  ValueKind kind;
  double d;
  string s;

  string render(int width) const;
}
```

Now, if we had never heard of inheritance (of if we were programming C, FORTRAN, Pascal, or any of the other pre-OO languages, we might have come up with an ADT something like this.\footnote{Actually, there are ways to make this more memory-efficient in those languages using constructs called “unions” or “variant records”, but they would not change the point of this example. }

- `kind` is a "control" data field.
  - It does not actually store useful data of its own.
  - It’s there to tell us which of the variants of value we have stored in any particular value.
  - We’ll use this mainly so that we can branch to code appropriate to that variant.

. . . .

Multi-Way Branching

```
string Value::render(int width) const {
  switch (kind) {
    case Numeric: {
      string buffer;
      ostringstream out (buffer);
      out << dv.d;
      return buffer.substr(0,width);
      }
    case String:
      return sv.s.substr(0.width);
    case Error:
      return string("** error **").substr(0.width);
  }
}
```

For example, to write the body for the `render` function, we would probably do something like this. First, we do a multi-way branch on the `kind` to get to the appropriate code for a numeric, string, or error value. Then in each branch we use a different technique to render the value as a string, then chop the string to the desired width.

- We can expect to see similar multi-way branches used to implement `clone()` or just about every other function we might write for manipulating `Values`.

. . . .

Variant Behavior under OO

```
class Value {
public:
  virtual string render(int width) const;
};

class NumericValue: public Value {
public:
  NumericValue (double dv);

  double d;

  string render(int width) const;
};

class StringValue: public Value {
public:
  StringValue (string sv);

  string s;

  string render(int width) const;
};

class ErrorValue: public Value {
  ErrorValue ();

  string render(int width) const;
};
```

Now, compare that with the OO approach.

- We represent each variant with a distinct subclass.
  - Only objects of the `NumericValue` class get the `d` data member.
  - Only objects of the `StringValue` class get the `s` data member.
  - None of them get the `kind` data member.
- This saves memory when we have large numbers of values floating about (as in a very large spreadsheet).
- But what's more important is how it affects the code we write for manipulating `Values`.

. . . .

Variants are Separated

```
string NumericValue::render(int width) const
{
  string buffer;
  ostringstream out (buffer);
  out << d;
  return buffer.substr(0,width);
}


string StringValue::render(int width) const {
  return s.substr(0.width);
}

string ErrorValue::render(int width) const {
  return string("** error **").substr(0.width);
}
```

Here's the OO take on the same `render` function.

- None of the details of how to render specific kinds of value have been changed.
- But we have repackaged that code into subclass-specific bodies.
  - The "variants" are now separate. In a team environment, different people can work on different variants separately.
  - Each subclass operation is simpler.
  - Most important of all, new kinds of values can be added without changing or recompiling the code of the earlier kinds of values.

. . . .

Suppose, for example, that we later decide to add to our spreadsheet the ability to manipulate dates. In the pre-OO style, we would need to look through all our code for any place we had a multi-way branch on the `kind`, then add a new branch with code to handle a date. Heaven help us if we miss one of those branches! If we're *lucky*, our code will crash during testing when we hit that branch with a date value. If we're unlucky, the crash occurs just as we're demo'ing the spreadsheet to upper management.

By contrast, to add dates in the OO style, we declare `DateValue`, a new subclass of `Value`. We write the code to render date values (which we would have to do in any case) and put it in its own `DateValue::render` body. Link it with the existing code, and we're ready to go. None of the existing code had to be touched at all.

Summary

We use inheritance in our programming designs whenever we find ourselves looking at objects that

- come in different varieties,
- each of which has its own slightly different way of doing the "same thing".

Conversely, if we *don't* see that kind of variant behavior, we probably have no need for inheritance.

- Should we have, for example, a subclass for `AlphaNumericStringValues`?
  - Unlikely – I can't think of a way in which the behavior of an alphanumeric string would vary, in the spreadsheet world, from that or an ordinary string.

. . . .

16.2 Using Inheritance

Using Inheritance

We'll look at 3 idioms describing good ways to use inheritance.

- Specialization
- Specification
- Extension

. . . .

16.2.1 Specialization

Specialization

When inheritance is used for *specialization*,

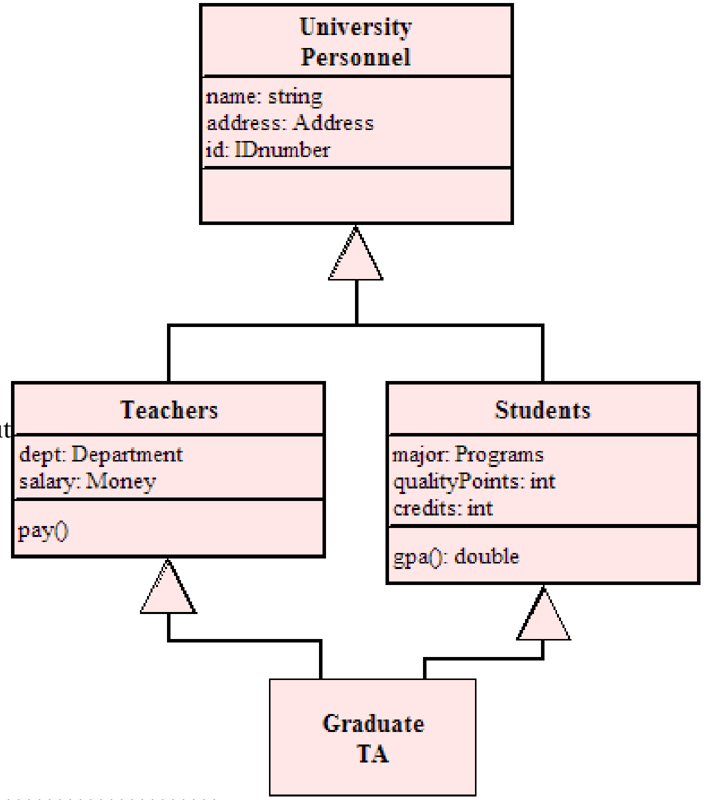
- The new class is a specialized form of the parent class
- but satisfies the specification of the parent in every respect.

  The new class may therefore be substituted for a value of the parent.

This is, in many ways, the "classical" view of inheritance.

. . . .

Recognizing Specialization



- A hallmark of specialization is that the base class
  - has objects of its own
  - may be processed in some applications without any contributions from the subclasses.

. . . .

Consider this hierarchy. It's quite likely that there are University Personnel who are neither teachers nor students (administrators, librarians, staff, etc.). As such, it's entirely likely that some applications (e.g., printing a University telephone directory) will work purely from the attributes and functions common to all University Personnel.

At the same time, there are variant behaviors among the subclasses that would prove useful in some applications. For example, Students have grade point averages (and probably other grade info not shown in this diagram) that would permit us to print grade reports and transcripts.

16.2.2 Specification

Specification

Inheritance for *specification*[1] takes place when

- a parent class specifies a common interface for all children
- but does not itself implement the behavior
  - Sometimes called the "shared protocol" pattern

. . . .

Defining Protocols

A *protocol* is a set of messages (functions) that can be used together to accomplish some desired task.

- The superclass defines the protocol.
- The subclasses implement the messages of the protocol in their own manner.
- Application code invokes the messages of the protocol without worrying about the individual methods.

. . . .

Recognizing the Specification Idiom of Inheritance

- Base class typically has no instances (objects)
  - Only objects are actually instances of subclasses
- Some operations may not even be implementable in the general base class

. . . .

[1] Yeah, it's a real pain that "specialization" and "specification" both look and sound so much alike. But I didn't make up these terms. And the words really don't mean anything like the same thing in their normal use, so just concentrate on what the words mean instead of what they sound like.

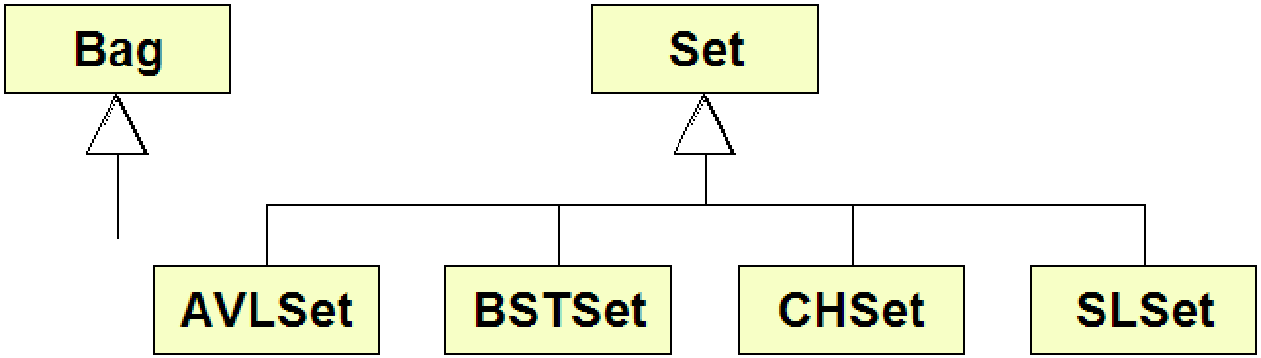
Example: Varying Data Structures

A common requirement in many libraries is to provide different data structures for the same abstraction.

- Allows application code writers to balance speed and storage requirements against expected size and usage patterns.
- Allows code in which the only mention of which data structure is being used comes when the actual objects are constructed

. . . .

libg++



For example, prior to the standardization of C++, the GNU g++ compiler was distributed with a library called `libg++`.

This library as extraordinary for the variety of implementations it provided for each major ADT. For example, the library declared a `Set` class that declared operations like adding an element to a set or checking the set to see if some value were a member of that set. However, the `Set` class itself contained no data structure that could actually hold any data elements.

Instead, a variety of subclasses of `Set` were provided. Each subclass implemented the required operations for being a Set, but provided its own data structure for storing and searching for the elements.

- `AVLSet` stored the data in AVL trees (a balanced binary tree)
- `BSTSet` stored the data in ordinary binary search trees

- CHSet stored the data in a conventional hash table
- SLSet stored the data in a singly-linked list

. . . .

In addition to these, there were subclasses for storing the elements in expandable arrays (similar to the std::vector), in dynamically expandable hash tables, in skip lists, etc.

The idea was that each of these data structures offered a slightly different trade off of speed versus storage, sometimes depending upon the size of the sets being manipulated.

Working with a Specialized Protocol

```
void generateSet (int numElements, Set& s, int *expected)
{
  int elements[MaxSetElement];


  for (int i = 0; i < MaxSetElement; i++)
    {
      elements[i] = i;
      expected[i] = 0;
    }


  // Now scramble the ordering of the elements array
  for (i = 0; i < MaxSetElement; i++)
    {
      int j = rand(MaxSetElement);
      int t = elements[i];
      elements[i] = elements[j];
      elements[j] = t;
    }
  // Insert the first numElements values into s
  s.clear();
  for (i = 0; i < numElements; i++)
    {
      s.add(elements[i]);
      expected[elements[i]] = 1;
    }
}
```

A programmer could write code, like the code shown here, that could work an *any* set.

- It is only in the code that *declared* a new set variable that an actual choice would have to be made.

. . . .

This meant that it was quite easy to write an application using one kind of set, measure the speed and storage performance, and then to change to a different variant of `Set` that would be a better match for the desired performance of the application.

`libg++` provided similar options, not only for sets, but also for bags (sets that permit duplicates, a.k.a. “multi-sets”), maps, and all manner of other container ADTs.

Abstract Base Classes

Adding to a Set Subclass

If we are working with `libg++` This is OK:

```
void foo (Set& s, int x)
{
   s.add(x);
   cout << x << " has been added." << endl;
}

int main ()
{
   BSTSet s;
   foo (s, 23);
    ⋮
```

. . . .

Adding to a General Set

But what should happen here?

```
void foo (Set& s, int x)
{
    s.add(x);
    cout << x << " has been added."  << endl;
}

int main ()
{
    Set s;
    foo (s, 23);
      ⋮
```

- `add()` makes no sense if `Set` doesn't have a data structure that can actually store elements.
  - In fact, `Set s;` makes no sense.

. . . .

How Do We Prevent This?

```
void foo (Set& s, int x)
{
    s.add(x);
    cout << x << " has been added."  << endl;
}
```

```
int main ()
{
    Set s;
    foo (s, 23);
      ⋮
```

- We could add a method so `Set()` that prints an error message when `add()` is called.
- Better is to force a proper choice of data structure at compile time.

. . . .

Abstract Member Functions

```
class Set {
      ⋮
    virtual Set& add (int) = 0;
      ⋮
};
```

- The `= 0` indicates that no method exists in this class for implementing this message.
- `add` is called an *abstract member function*.
  - Subclasses must provide the actual methods (bodies) for these functions.

. . . .

Abstract Classes

An *abstract class* in C++ is any class that

- contains an `= 0` annotation on a member function, or

- inherits such a function and does not provide a method for it.

"Abstract classes" are also known as *pure virtual classes*.

. . . .

Set as an Abstract Class

`Set` in `libg++` is a good example of a class that should be abstract.

- We can't possibly implement `Set::Add`
  - we need to do that in its subclasses.

. . . .

Limitations of Abstract Classes

Abstract classes carry some limitations, designed to make sure we use them in a safe manner.

```
class Set {
   ⋮
  virtual Set& add (int) = 0;
   ⋮
};
  ⋮
void foo (Set& s, int x) // OK
  ⋮

int main () {
   Set s;   // error!
   foo (s, 23);
     ⋮
```

- You cannot construct an object whose type is an abstract class.
- You cannot declare function parameters of an abstract class type when passing parameters "by copy".

  - but you can pass pointers/references to the abstract class type.

. . . .

Abstract Classes & Specification

- Base classes in inheritance-for-specification are often abstract
  - Base class exists to define a protocol
  - Not to provide actual objects
- Our `value.h` spreadsheet *Value* class is abstract.

  We cannot expect, for example, to write a `render` function that would work for all values. We have to rely on the subclasses to provide the code for rendering themselves.
- So is our spreadsheet `expression.h` *Expression* class.

  Look at the declaration and see which functions are marked as abstract. Ask yourself if you could implement any of those for all possible expressions. For example, can you write a rule to `evaluate` any expression? No. Can you write a rule to evaluate a `PlusNode`? Or perhaps a `NumericConstant`? Those seem much more plausible.

. . . .

16.2.3 Extension

Extension

In this style of inheritance, a limited number of "new" abilities is grafted onto an otherwise unchanged superclass.

```
class FlashingString: public StringValue {
  bool _flash;
public:
  FlashingString (std::string);
  void flash();
  void stopFlashing();
};
```

. . . .

Are Extensions OK?

- Extensions are often criticized as "hacks" reflecting afterthoughts & poor hierarchy designs.
- There is often a cleaner way to achieve the same design
  - A "socially acceptable" form of extension is the *mixin*

. . . .

Mixins

A mixin is a class that makes little sense by itself, but provides a specialized capability when used as a base class.

- Mixins in C++ often wind up involving multiple inheritance.

. . . .

Mixin Example: Noisy

Noisy is a mixin I use when debugging.

```
#ifndef NOISY_H
#define NOISY_H

class Noisy
{
  static int lastID;
  int id;
public:
  Noisy();
  Noisy(const Noisy&);
  virtual ~Noisy();

  void operator= (const Noisy&);
};

#endif
```

```
#include "noisy.h"
#include <iostream>

using namespace std;

int Noisy::lastID = 0;

Noisy::Noisy()
{
  id = lastID++;
  cerr << "Created object " << id << endl;
}

Noisy::Noisy(const Noisy& x)
{
  id = lastID++;
  cerr << "Copied object " << id << " from object " << x.id << endl;
}

Noisy::~Noisy()
{
  cerr << "Destroyed object " << id << endl;
}


void Noisy::operator= (const Noisy&) {}
```

- Keep track of when and where constructors and destructors are being invoked
- Also helps to catch excessive copying.

. . . .

Using Noisy

```
class TreeSet
  : public Set, public Noisy
{
    ⋮
```

- This particular mixin also works if used as a data member.
  - Should we worry about multiple inheritance conflicts?
    - Probably not, because `Noisy` declares so few public members it is unlikely to conflict with the “real” member names of my classes.

. . . .

Mixin Example: checking for memory leaks

```
#ifndef COUNTED_H
#define COUNTED_H
```

```
class Counted
{
  static int numCreated;
  static int numDestroyed;
public:
  Counted();
  Counted(const Counted&);
  virtual ~Counted();

  static void report();

};

#endif
```

```
#include "counted.h"
#include <iostream>

using namespace std;

int Counted::numCreated = 0;
int Counted::numDestroyed = 0;

Counted::Counted()
{
  ++numCreated;
}

Counted::Counted(const Counted& x)
{
  ++numCreated;
}

Counted::~Counted()
{
  ++numDestroyed;
}



void Counted::report()
{
  cerr << "Created " << numCreated << " objects." << endl;
  cerr << "Destroyed " << numDestroyed << " objects." << endl;
}
```

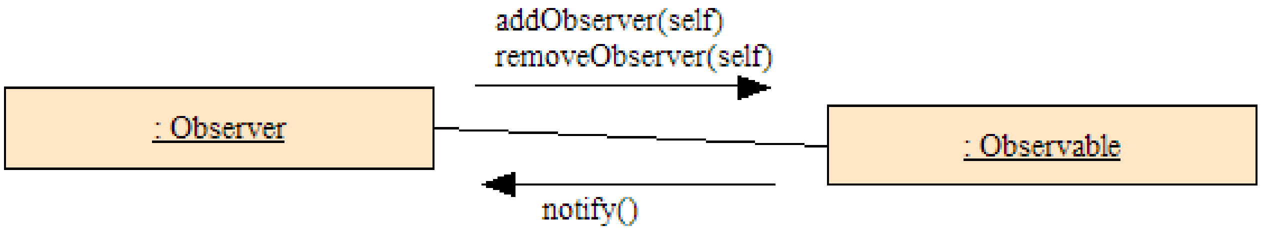
- A similar example is offered by the Counted class.

- Lets me see if I have created more objects than I have destroyed.

. . . .

16.3 The Observer Pattern

The Observer Pattern



- A "design pattern" or idiom of OO programming
- Based on 2 mixins
  - Observer
    - can ask an Observable object for notification of any changes
  - Observable
    - will keep track of a list of Observers and notify them when its state changes

```
#ifndef OBSERVER_H
#define OBSERVER_H

//
// An Observer can register itself with any Observable object
// cell by calling the obervable's addObserver() function. Subsequently,
// the oberver wil be notified whenever the oberver calls its
// notifyObservers() function (usually whenever the obervable object's
// value has changed.
//
// Notification occurs by calling the notify() function declared here.
```

```
class Observable;

class Observer
{
public:
  virtual void notify (Observable* changedObject) = 0;
};
#endif
```

Here is an `Observer` mixin. Note that there's not a whole lot to it. `Observable` need to be able to `notify` observers.

Here is the Java equivalent. The Observer/Observable pattern is so common that it is part of the standard Java API, and has been for quite some time. It has some minor differences. The function is called "update" instead of "notify" and can take a second parameter, but the basic idea is the same.

```
#ifndef OBSERVABLE_H
#define OBSERVABLE_H

#include "observerptrseq.h"

// An Observable object allows any number of Observers to register
// with it. When a significant change has occured to the Observable object,
// it calls notifyObservers() and each registered observer will be notified.
// (See also oberver.h)

class Observer;

class Observable
{
public:

  // Add and remove observers
  void addObserver (Observer* observer);
  void removeObserver (Observer* observer);

  //   For each registered Observer, call notify(this)
  void notifyObservers();

private:
  ObserverPtrSequence observers;
};



#endif
```

`Observable` is not much more complicated. It has functions allowing an observer to register itself for future notifications, and a utility function that the observable object calls to talk its current list of observers and notify each of them.

```
#include "observable.h"
#include "observer.h"



// An Observable object allows any number of Observers to register
// with it. When a significant change has occured to the Observable object,
// it calls notifyObservers() and each registered observer will be notified.
//

// Add and remove observers
void Observable::addObserver (Observer* observer)
{
  observers.addToFront (observer);
}



void Observable::removeObserver (Observer* observer)
```

```
{
  ObserverPtrSequence::Position p = observers.find(observer);
  if (p != 0)
    observers.remove(p);
}

//   For each registered Observer, call hasChanged(this)
void Observable::notifyObservers()
{
  for (ObserverPtrSequence::Position p = observers.front();
       p != 0; p = observers.getNext(p))
    {
      observers.at(p)->notify(this);
    }
}
```

The implementation details are above.

The Java version of Observable is similar.

. . . .

16.3.1 Applications of Observer

Example: Propagating Changes in a Spreadsheet

Example: Propagating Changes in a Spreadsheet

Anyone who has used a spreadsheet has observed the way that, when one cell changes value, all the cells that mention that first cell in their formulas change, then all the cells the mention those cells change, and so on, in a characteristic "ripple" effect until all the effects of the original change have played out.

There are several ways to program that effect. One of the more elegant is to use the Observer/Observable pattern.

. . . .

Cells Observe Each Other

```
class Cell: public Observable, Observer
{
public:
```

The idea is that cells will observe one another.

- Suppose cell B2 contains the formula “2*A1 + B1”.
  - Then B2 will observe A1 and B1.

    It will use the `Observable::addObserver` function to add itself as an observer of both those cells.
  - If something were to happen to A1 that changes its value (e.g., we click on A1 and then enter a new value), then
    - A1 will call `notifyObservers()`, which goes through its list of observers, including, eventually including B2, notifying each one.
    - When B2 is notified, it can re-evaluate its formula, change its value, and notify **its** observers.

. . . .

Changing a Cell Formula

```
void Cell::putFormula(Expression* e)
{
  if (theFormula != 0)
    {  ❶
      CellNameSequence oldReferences = theFormula->collectReferences();
      for (CellNameSequence::Position p = oldReferences.front();
           p != 0; p = oldReferences.getNext(p))
        {
          Cell* c = theSheet.getCell(oldReferences.at(p));
          if (c != 0)
```

```
            c->removeObserver (this);
        }
      delete theFormula;
    }
  theFormula = e; ❷
  if (e != 0)
    {     ❸
      CellNameSequence newReferences = e->collectReferences();
      for (CellNameSequence::Position p = newReferences.front();
           p != 0; p = newReferences.getNext(p))
        {
          Cell* c = theSheet.getCell(newReferences.at(p));
          if (c != 0)
            c->addObserver (this);
        }
    }
  theSheet.cellHasNewFormula (this); ❸
}
```

Here's the code that's actually invoked to change the expression stored in a cell.

❶ The first if statement, and it's loop, looks at the expression already stored in the cell. It loops through all cells already named in the expression and tells them that this cell is no longer observing them (`removeObserver`).

❷ After that, we store the new expression (`e`) into the cell (`theFormula` is a data member of `Cell`).

❸ The next if statement and the loop inside look almost like the first one. But now we are looking at the new formula, and calling `addObserver` instead. So now any cells mentioned in the new expression will notify this one when their values change.

❹ At this point, we have set up the network of cells observing other cells. This particular cell has been given a new expression, so there's a good change its value will change once we evaluate that expression. For technical reasons, we don't do so immediately. Instead, we tell the spreadsheet that this cell has a new formula. The spreadsheet keeps a queue of cells that need to be re-evaluated, and processes them one at a time.

. . . .

Evaluating a Cell's Formula

```
const Value* Cell::evaluateFormula()
{
  Value* newValue = (theFormula == 0)
    ? new StringValue()
    : theFormula->evaluate(theSheet);   ❶

  if (theValue != 0 && *newValue == *theValue) ❷
    delete newValue;    ❸
  else
    {    ❹
      delete theValue;
      theValue = newValue;
      notifyObservers();   ❺
    }
  return theValue;
}
```

Eventually the spreadsheets calls this function on our recently changed cell.

❶ We start by checking to see if the formula is not null. If it is not, we evaluate it to get the value of the new expression, `newValue`.

❷ We make sure the cell's old value (stored in the cell's data member `theValue`) is not null, then check to see if it is equal to the new value.

❸ If they are equal, we don't need the new value and can throw it away.

❹ If they are not equal, we throw out the old value and save the new one. Now our cell's value has definitely changed.

❺ So what do we do? We notify our observers.

. . . .

Notifying a Cell's Observers

```
void Cell::notify (Observable* changedCell)
{
  theSheet.cellRequiresEvaluation (this);
}
```

What does an observing cell do when it is notified? It tells the spreadsheet that it needs to be re-evaluated.

- Again, the spreadsheet puts the cell into a queue, but eventually calls `evaluateFormula` on that cell,
  - which may change value and notify *its* observers,
  - which will tell the spreadsheet that they need to be re-evaluated,
  - which will eventually call `evaluateFormula` on them,

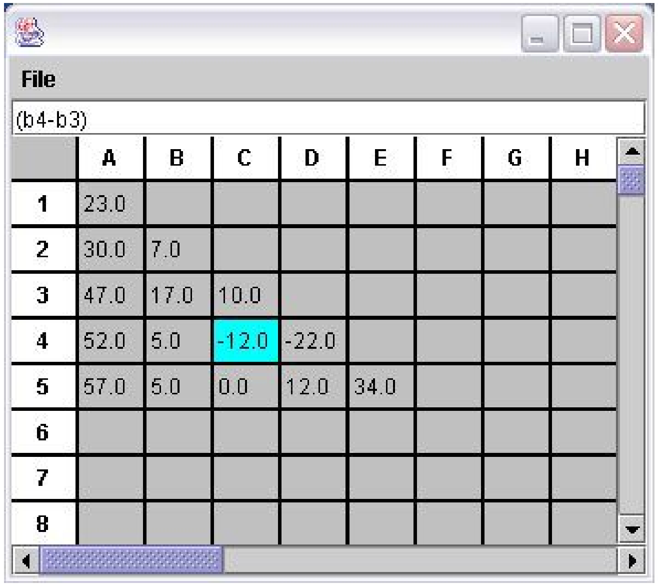
and so on.

Eventually the propagation trickles to an end, as we eventually re-evaluate cells that either do not change value or that are not themselves mentioned in the formulae of any other cells.

. . . .

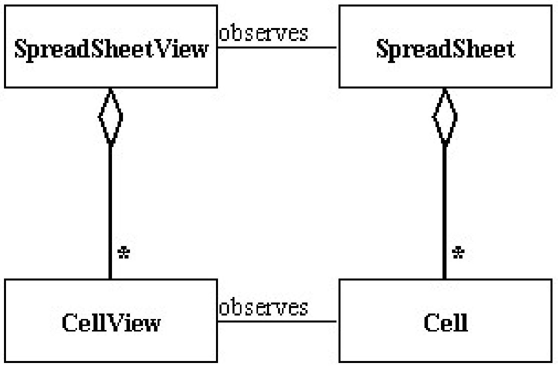
Example: Observer and GUIs

Example: Observer and GUIs



(b4-b3)

	A	B	C	D	E	F	G	H
1	23.0							
2	30.0	7.0						
3	47.0	17.0	10.0					
4	52.0	5.0	-12.0	-22.0				
5	57.0	5.0	0.0	12.0	34.0			
6								
7								
8								



cellView[0,0] observes ssheet.getCell("a1")

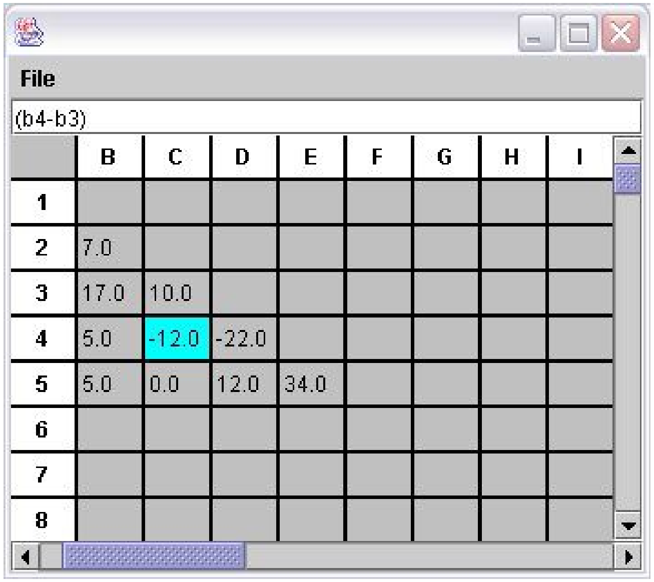
cellView[0,1] observes ssheet.getCell("a2")

A spreadsheet GUI contains a rectangular array of `CellViews`. Each `CellView` observes one `Cell`

- When value in a cell changes, it notifies its observers, as we have already seen. Some of those observers are other cells, as described earlier. But one of those observers might be a `CellView`.
- The observing `CellView` then redraws itself with the new value

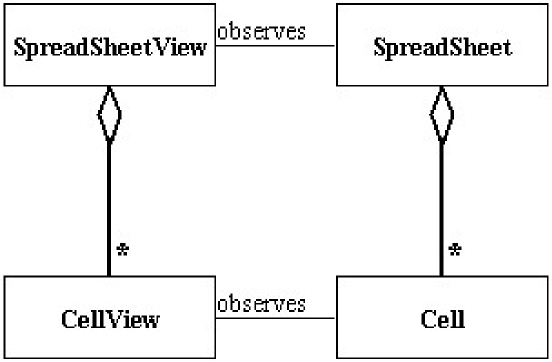
. . . .

Scrolling the Spreadsheet



(b4-b3)

	B	C	D	E	F	G	H	I
1								
2	7.0							
3	17.0	10.0						
4	5.0	-12.0	-22.0					
5	5.0	0.0	12.0	34.0				
6								
7								
8								



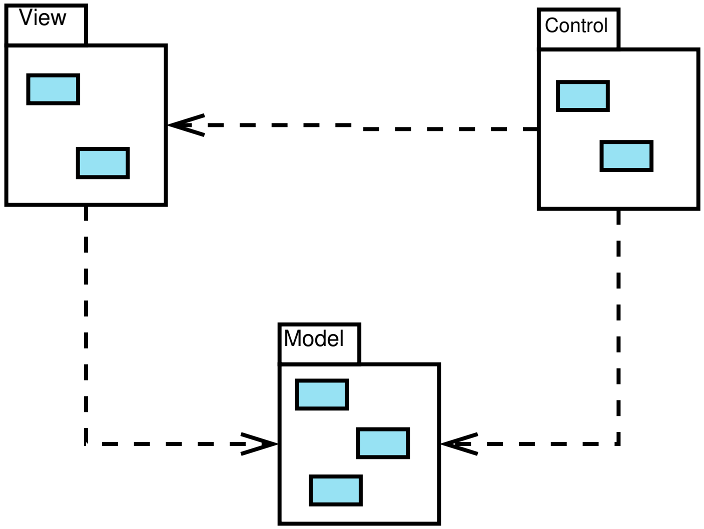
```
cellView[0,0] observes ssheet.getCell("b1")
cellView[0,1] observes ssheet.getCell("b2")
```

Not every cell will have a `CellView` observer.

- That's because most of the cells in a spreadsheet are not actually visible at a given time.

- Whenever we scroll the spreadsheet GUI, the `CellViews` each register themselves with a different cell, depending on how far we have scrolled.

Model-View-Controller (MVC) Pattern



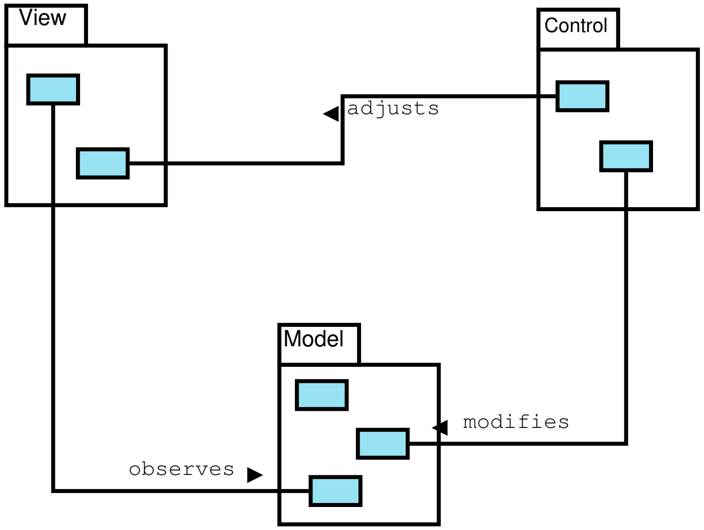
- A powerful approach to GUI construction
- Separate the system in 3 subsystems
  - Model: the "real" data managed by the program
  - View: portrayal of the model
    - updated as the data changes
  - Controller: input & interaction handlers
- View observes the Model

This pattern has many advantages. GUI code is hard to test. Keeping the core data independent of the GUI means we can test it using unit or similar "easy" techniques. We can also change the GUI entirely without altering the core classes. For example, my implementation of the spreadsheet has both a graphic form (as shown in the prior section) and a text-only interface that can be used over a simple telnet connection.

Separating the control code means that we can test the view by driving it from a custom Control that simply issues a pre-scripted set of calls on the view and model classes

. . . .

MVC Interactions



How do we actually accomplish this?

- The Oberver/Observable pattern is one important component of the MVC.
  - It allows the Model classes to notify appropriate parts of the GUI without knowing anything detailed about the GUI class interfaces.

. . . .

16.4 Abusing Inheritance

Just as there are idioms for good uses of inheritance, there are some that programmers have tried, and that new programmers often “rediscover”, that are really not advisable.

16.4.1 Construction

Occurs when a subclass is created that uses the superclass as the implementing data structure

- May violate the is-a relationship normally indicated by inheritance.

```
struct Cell {
   int    data;
   Cell* next;

   Cell (int i) {data = i; next = this;}
   Cell (int i, Cell* n)
             {data = i; next = n};
};
```

Here is an example, drawn from a textbook that shall remain anonymous. This cell (no relation to the notion of a cell in a spreadsheet) class defines a "utility" linked list node.

```
class CircularList {
    Cell* rear;
public:
    CircularList();
    bool empty();
    void addFront (int);
    int removeFront ();
    int addRear (int);
};
```

We can use this to create a circular list (a linked list in which the last node in the list points back to the first node). Circular lists are useful in that they support efficient adding at either end and removing from one end.

```
class Queue: public CircularList {
public:
    Queue();
    void enterq (int x) { addRear(x);}
    int leaveq()        {removeFront();}
    // inherit bool empty();
};


class Stack: public CircularList {
public:
    Stack();
    void push (int x) {addFront(x);}
    void pop() {removeFront();}
    // inherit bool empty()
};
```

This list could be used to implement stacks and queues.
So we can write applications like the following:

```
    Queue q;
    q.enterq (23);
    q.enterq (42);
    q.leaveq ();
    bool b = q.empty();
    q.addFront (21);   // oops!
```

The last call is a problem. We should not be able to add to the front of a queue.

- Public inheritance makes too much additional interface visible

– need to hide the underlying operations.

Private Inheritance

```
class Queue: private CircularList {
public:
   Queue();
   void enterq (int x) { addRear(x);}
   int leaveq()        {removeFront();}
   CircularList::empty;
};


class Stack: private CircularList {
public:
   Stack();
   void push (int x) {addFront(x);}
   void pop() {removeFront();}
   CircularList::empty;
};
```

It is possible to solve this problem by private inheritance. We inherit, but all inherited members are private in the subclass.

Private Inheritance

Private inheritance hides inherited members from application code.

- It can be used to share data structure and methods without affecting the interface
- Contrast this with public inheritance, where the major motivation is to share protocol/interface.

What's Wrong with This?

- With private inheritance (or judicious use of protected members), inheritance for the purpose of code sharing is possible,

- But
  - the resulting inheritance hierarchy is often counter-intuitive and may clash with other inheritance motivations.
  - it is usually easier to do the same thing using aggregation/composition rather than inheritance:

```
class Queue: {
   CircularList c;
public:
   Queue();
   void enterq (int x) { c.addRear(x);}
   int leaveq()        {c.removeFront();}
   bool empty()        {return c.empty();}
};
```

Doesn't this just seem simpler?

- With inline function expansion, this need not be any slower than the inherited versions.

16.4.2 Subset != Subclass

- We often say that inheritance captures an "is a" relationship.
  - But "is a" is vague and subject to many interpretations.
  - A common interpretation is "is a specialized form of".

A Bird in the Hand

```
class Animal;
class Bird: public Animal { ...
class BlueJay: public Bird { ...
```

- Seems logical, right?

is Worth 2 in the Sky?

Let's postulate some members for Birds:

```
class Bird: public Animal {
   Bird();
   double altitude() const;

   void fly();
      // post-condition: altitude() > 0.
     ⋮
};

class BlueJay: public Bird
{
    ⋮
```

Is an Ostrich a-kind-of Bird?

```
class Ostrich: public Bird
{
  Ostrich();
  // Inherits
  // double altitude() const;

  // void fly();
  //post-condition: altitude() > 0.
```

- Now, Ostrich can provide its own method for fly()

```
void Ostrich::fly()
//post-condition: altitude() > 0.
{
  plummet();
}
```

- but it can’t satisfy that post-condition.
  - An unpleasant surprise for someone traversing a list of Bird’s.

Substitutability

The ostrich/bird hierarchy violates the substitutability principle:

- Class A should be a public subclass of B only if a value of class A can logically be substituted wherever a value of type B is expected.
  - Instead of is-a, perhaps we should say that public inheritance captures an "observes the protocol of" relation.
- You can view this dilemma as a failure of analysis:
  - Who said all birds could fly?
  - A FlyingBirds subclass of Bird would clarify the situation.
- Anyone want to tackle a platypus?

Fixing a Broken Hierarchy

When two classes share code and data but do not share an is-a (protocol) relationship.

- Mouse & Tablet
- Stack& Queue

- Better to factor shared code and data into a common parent class
  - e.g., Pointing Device
- or factor shared code and data into a common implementation structure, accessed via composition
  - e.g., implement Stack & Queue via a LinkedList data member

16.5 value.h

```
#ifndef VALUE_H
#define VALUE_H

#include <string>
#include <typeinfo>

//
// Represents a value that might be obtained for some spreadsheet cell
// when its formula was evaluated.
//
// Values may come in many forms. At the very least, we can expect that
// our spreadsheet will support numeric and string values, and will
// probably need an "error" or "invalid" value type as well. Later we may
// want to add addiitonal value kinds, such as currency or dates.
//
class Value
{
public:
  virtual ~Value() {}
  virtual std::string render (unsigned maxWidth) const = 0;
  // Produce a string denoting this value such that the
  // string's length() <= maxWidth (assuming maxWidth > 0)
  // If maxWidth==0, then the output string may be arbitrarily long.
  // This function is intended to supply the text for display in the
  // cells of a spreadsheet.


  virtual Value* clone() const = 0;
  // make a copy of this value

protected:
  virtual bool isEqual (const Value& v) const = 0;
  //pre: typeid(*this) == typeid(v)
  //  Returns true iff this value is equal to v, using a comparison
  //  appropriate to the kind of value.

  friend bool operator== (const Value&, const Value&);
};

inline
bool operator== (const Value& left, const Value& right)
{
  return (typeid(left) == typeid(right))
    && left.isEqual(right);
}

#endif
```

16.6 expression.h

```
#ifndef EXPRESSION_H
#define EXPRESSION_H

#include <string>
#include <iostream>

#include "cellnameseq.h"

class SpreadSheet;
class Value;

// Expressions can be thought of as trees.  Each non-leaf node of the tree
// contains an operator, and the children of that node are the subexpressions
// (operands) that the operator operates upon.  Constants, cell references,
// and the like form the leaves of the tree.
//
// For example, the expression (a2 + 2) * c26 is equivalent to the tree:
//
//                 *
//                /
//               +   c26
//              /
//             a2   2

class Expression
{
public:
```

```
virtual ~Expression() {}


// How many operands does this expression node have?
virtual unsigned arity() const = 0;

// Get the k_th operand
virtual const Expression* operand(unsigned k) const = 0;
//pre: k < arity()




// Evaluate this expression
virtual Value* evaluate(const SpreadSheet&) const = 0;



// Copy this expression (deep copy), altering any cell references
// by the indicated offsets except where the row or column is "fixed"
// by a preceding $. E.g., if e is  2*D4+C$2/$A$1, then
// e.copy(1,2) is 2*E6+D$2/$A$1, e.copy(-1,4) is 2*C8+B$2/$A$1
virtual Expression* clone (int colOffset, int rowOffset) const = 0;



virtual CellNameSequence collectReferences() const;


static Expression* get (std::istream& in, char terminator);
```

```
  static Expression* get (const std::string& in);
  virtual void put (std::ostream& out) const;




  // The following control how the expression gets printed by
  // the default implementation of put(ostream&)

  virtual bool isInline() const = 0;
  // if false, print as functionName(comma-separated-list)
  // if true, print in inline form

  virtual int precedence() const = 0;
  // Parentheses are placed around an expression whenever its precedence
  // is lower than the precedence of an operator (expression) applied to it.
  // E.g., * has higher precedence than +, so we print 3*(a1+1) but not
  // (3*a1)+1

  virtual string getOperator() const = 0;
  // Returns the name of the operator for printing purposes.
  // For constants, this is the string version of the constant value.




};




inline std::istream& operator>> (std::istream& in, Expression*& e)
{
  string line;
  getline(in, line);
  e = Expression::get (line);
  return in;
}


inline std::ostream& operator<< (std::ostream& out, const Expression& e)
{
  e.put (out);
  return out;
}

inline std::ostream& operator<< (std::ostream& out, const Expression* e)
{
  e->put (out);
  return out;
}

#endif
```

Chapter 17

Sharing Pointers and Garbage Collection

Swearing by Sharing

We've talked a lot about using pointers to share information, but mainly as something that causes problems.

- We have a pretty good idea of how to handle ourselves when we have pointers among our data members and *don't* want to share.

  In that case, we rely on implementing our own deep copying so that every "container" has distinct copies of all of its components.

  ```
  class Catalog {
     ⋮
     Catalog (const Catalog& c);
     Catalog& operator= (const Catalog& c);
     ~Catalog();
     ⋮
  private:
    Book* allBooks;      // array of books
    int numBooks;
  ```

```
};


Catalog::Catalog (const Catalog& c)
: numBooks(c.numBooks)
{
   allBooks = new Book[numBooks];
   copy (c.allBooks, c.allBooks+numBooks, allBoooks);
}

Catalog& Catalog::operator= (const Catalog& c)
{
  if (*this != c)
    {
      delete [] allBooks;
      numBooks = c.numBooks;
      allBooks = new Book[numBooks];
      copy (c.allBooks, c.allBooks+numBooks, allBoooks);
    }
  return *this;
}

Catalog::~Catalog()
{
  delete [] allBooks;
}
```

For some data structures, this is OK. If we are using a pointer mainly to give us access to a dynamically allocated array, we can copy the entire array as necessary. In the example shown here, we would want each catalog to get its own distinct array of books. So we would implement a deep copy for the assignment operator and copy constructor, and delete the allBooks pointer in the destructor.

- But not every data structure can be treated this way.

- Sometimes, sharing is essential to the behavior of the data structure that we want to implement.
- (In data structures, there is an entire class of structures and algorithms associated with “graphs” that exhibit this property.)

. . . .

17.1 Shared Structures

Shared Structures

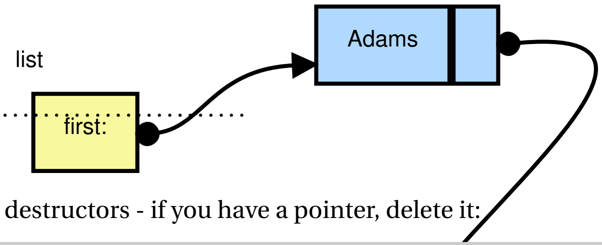
In this section, we will introduce three examples that we will explore further in the remainder of the lesson. All three involve some degree of essential sharing.

. . . .

17.1.1 Singly Linked Lists

Singly Linked Lists

We'll start with a fairly prosaic example. In its simplest form, a singly linked list involves no sharing, and so we could safely treat all of its components as deep-copied.



SLL Destructors

In particular, we can take a simple approach of writing the destructors - if you have a pointer, delete it:

```
struct SLNode {
   string data;
   SLNode* next;
     ⋮
   ~SLNode () {delete next;}
};

class List {
   SLNode* first;
public:
     ⋮
   ~List() {delete first;}
};
```

Problem: stack size is $O(N)$ where N is the length of the list.

If a *List* object gets destroyed, its destructor will delete its *first* pointer. That node (Adams in the picture) will have its destructor called as part of the delete, and it will delete its pointer to Baker. The Baker node's destructor will delete the pointer to Davis. At then end, we have successfully recovered all on-heap memory (the nodes) with no problems.

Now, this isn't really ideal. At the time the destructor for Davis is called, there are still partially executed function activations for the Baker and Adams destructors and for the list's destructor still on the call stack, waiting to finish. That's no big deal with only three nodes in the list, but if we had a list of, say, 10000 nodes, then we might not have enough stack space for 10000 uncompleted calls. So, typically, we would actually use a more aggressive approach with the list itself:

Destroy the List, not the Nodes

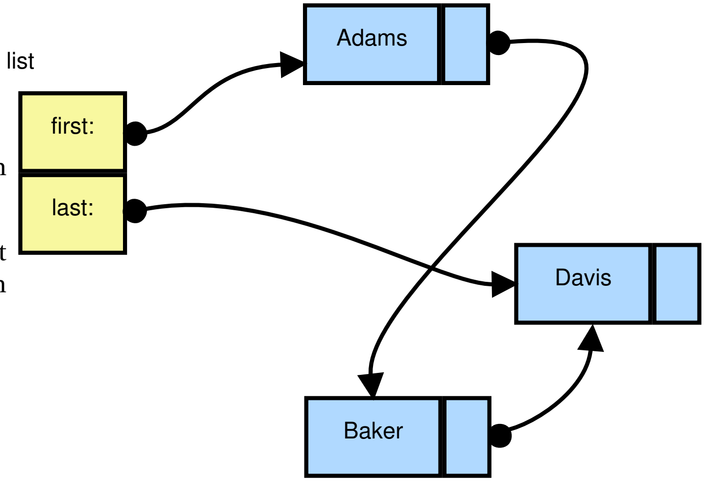
```
struct SLNode {
   string data;
   SLNode* next;
     ⋮
   ~SLNode () {/* do nothing */}
};

class List {
   SLNode* first;
public:
     ⋮
   ~List()
    {
      while (first != 0)
        {
          SLNode* next = first->next;
          delete first;
          first = next;
        }
    }
};
```

This avoids stacking up large numbers of recursive calls.

. . . .

But, if we weren't worried about stack size, then our first approach will be fine.

First-Last Headers



But now let's consider one of the more common variations on linked lists.

- If our header contains pointers to both the first and last nodes of this list, then we can do O(1) insertions at both ends of this list.

- Notice, however, that the final node in the list is now "shared" by both the list header and the next-to-last node.

. . . .

FLH SLL Destructor

So, if we were to extend our basic approach of writing destructors that simply delete their pointers:

```
struct SLNode {
   string data;
   SLNode* next;
     ⋮
   ~SLNode () {delete next;}
};

class List {
   SLNode* first;
   SLNode* last;
public:
      ⋮
   ~List() {delete first; delete last;}
};
```

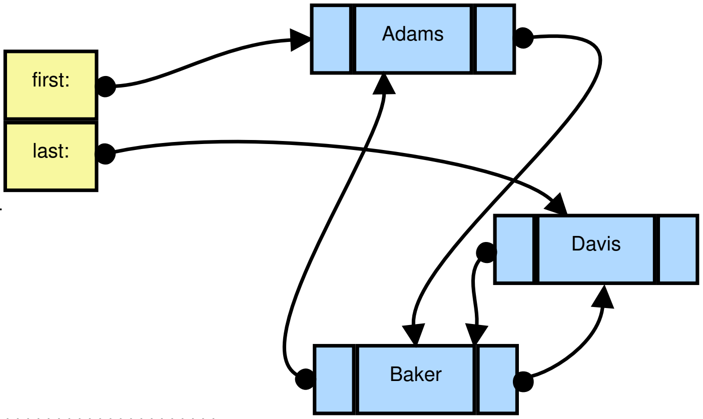
Then, when a list object is destroyed, the final node in the list will actually be deleted twice. Deleting the same block of memory twice can corrupt the heap (by breaking the structure of the free list) and eventually cause the program to fail.

. . . .

17.1.2 Doubly Linked Lists

Doubly Linked Lists

Now, let's make things just a little *more* difficult.
If we consider doubly linked lists, our straightforward approach of "delete everything" is really going to be a problem.



. . . .

DLL Aggressive Deleting

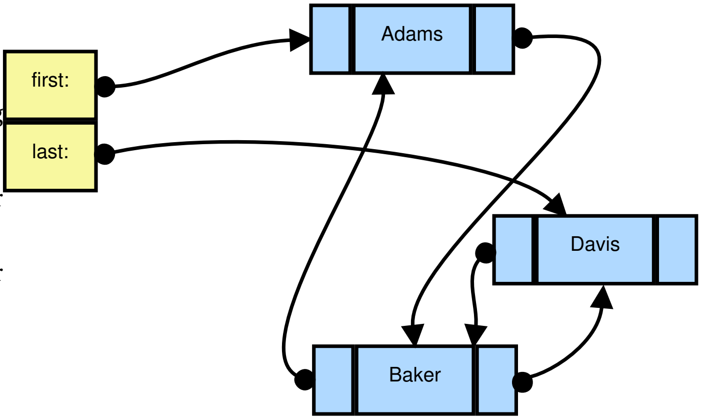
```
struct DLNode {
   string data;
   DLNode* prev;
   DLNode* next;
```

```
      ⋮
   ~DLNode () {delete prev; delete next;}
};

class List {
   DLNode* first;
   DLNode* last;
public:
      ⋮
   ~List() {delete first; delete last;}
};
```

. . . .

Deleting the DLL

- When a list object is destroyed, it will start by deleting the *first* pointer.
  - That node (Adams) will delete its *next* pointer (pointing to Baker).
  - That second node will delete its *prev* pointer (Adams).



- Now we've deleted the same node twice, potentially corrupting the heap.
  - But, worse, the Adam node's destructor will be invoked again.
  - It will delete its *next* pointer, and we will have deleted the Baker node a second time.

- Then the Baker node deletes its *prev* pointer *again*.

. . . .

Deleting and Cycles

We're now in an infinite recursion,

- which will continue running until either the heap is so badly corrupted that we crash when trying to process a delete,
- or when we finally fill up the activati0on stack to its maximum possible size.

What makes this so much nastier than the singly linked list?

- It's the fact that not only are we doing sharing via pointers, but that the various connections form *cycles*, in which we can trace a path via pointers from some object eventually back to itself.

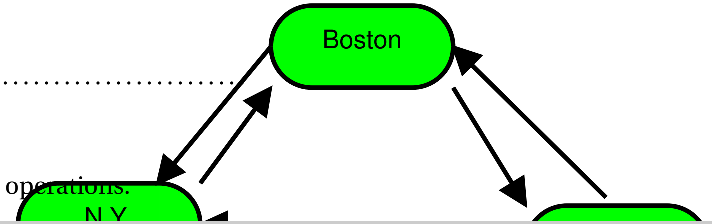
. . . .

17.1.3 Airline Connections

Airline Connections

Lest you think that this issue only arises in low-level data structures, let's consider how it might arise in programming at the application level.

This graph illustrates flight connections available from an airline.



. . . .

Aggressively Deleting a Graph

If we were to implement this airport graph with Big 3-style operations:

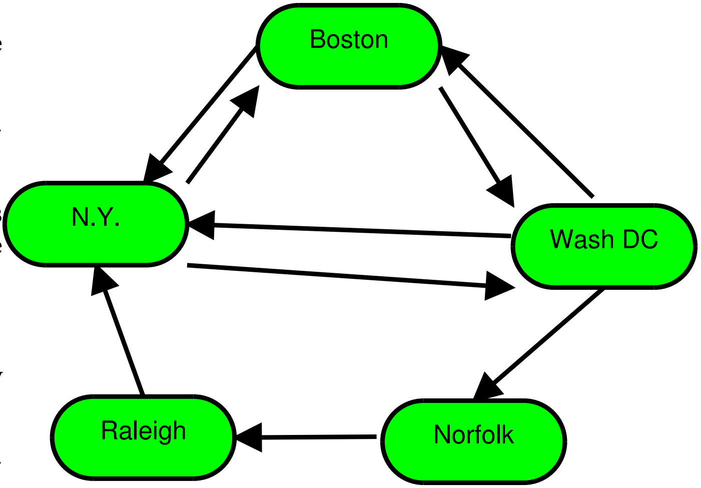
```
class Airport
{
   ⋮

private:
   vector<Airport*> hasFlightsTo;
};

Airport::~Airport()
{
   for (int i = 0; i < hasFlightsTo.size(); ++i)
      delete hasFlightsTo[i];
}
```

we would quickly run into a disaster.

. . . .

Deleting the Graph

Suppose that we delete the Boston airport.



- Its destructor would be invoked, which would delete the N.Y. airport and Wash DC airports.
  - Let's say, for the sake of example, that the NY airport is deleted first.
- The act of deleting the pointer to the NY airport causes its destructor to be invoked, which would delete the Boston and Wash DC airports.
  - But Boston has already been deleted.
    - If we don't crash right away, we will quickly wind up deleting Wash DC twice.
- In fact, we would wind up, again, in an infinite recursion among the destructors.

. . . . This should not be a big surprise. Looking at the graph, we can see that it is possible to form cycles. (In fact, if there is any node in this graph that *doesn't* participate in a cycle, there would be something very wrong with our airline. Either we would have planes piling up at some airport, unable to leave; or we would have airports that run out of planes and can't support any outgoing flights.

The Airline

Now, you might wonder just how or why we would have deleted that Boston pointer in the first place. So, let's add a bit of context.

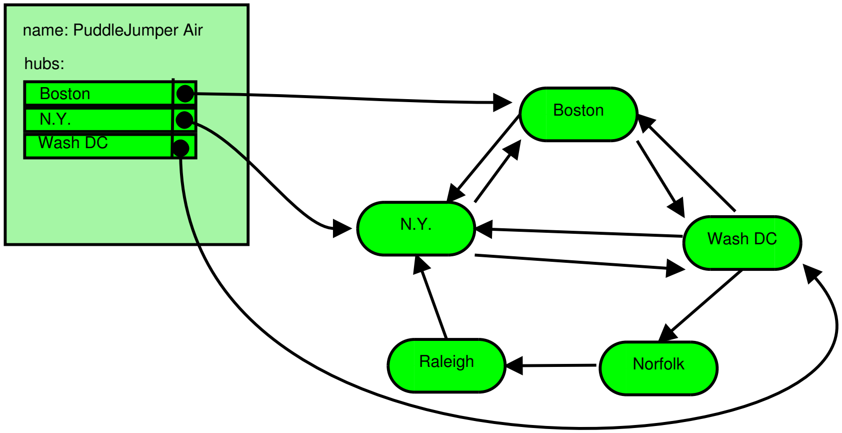
• The airport graph is really a part of the description of an airline:

```
class AirLine {
   ⋮
   string name;
   map<string, Airport*> hubs;
};
```

```
AirLine::~Airline()
{
  for (map<string, Airport*>::iterator i = hubs.begin;
       i != hubs.end(); ++i)
    delete i->second;
}
```

. . . .

The AirLine Structure



- The map *hubs* provides access to all those airports where planes are serviced and stored when not in flight, indexed by the airport name.
  - Not all airports are hubs.
- An airport that is not a hub is simply one where planes touch down and pick and discharge passengers while on their way to another hub.

Suppose that PuddleJumper Air goes bankrupt.

- It makes sense that when an airline object is destroyed, we would delete those hub pointers.
  - But we've seen that this is dangerous.

. . . .

Can We Do Better?

Now, that's a problem. But what makes this example particularly vexing is that it's not all that obvious what would constitute a better approach.

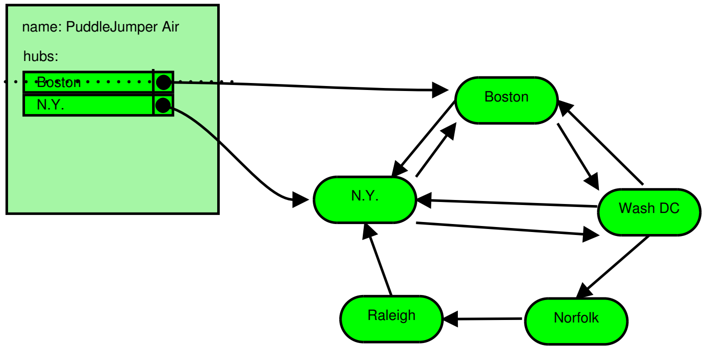
- Let's consider some other changes to the airline structure.

. . . .

Changing the Hubs

Suppose that Wash DC were to lose its status as a hub.

Even though the pointer to it were removed from the *hubs* table, the Wash DC airport needs to remain in the map.

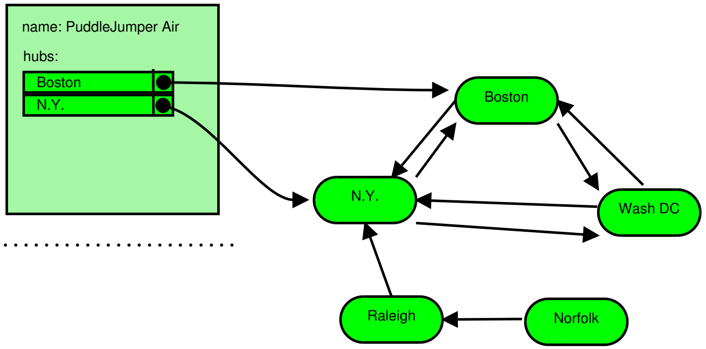


. . . .

Changing the Connections

On the other hand, if Wash DC were to drop its service to Norfolk, one might argue that Norfolk and Raleigh should then be deleted, as there would be no way to reach them.

- But how could you write code into your destructors and your other code that adds and removes pointers that could tell the difference between these two cases?

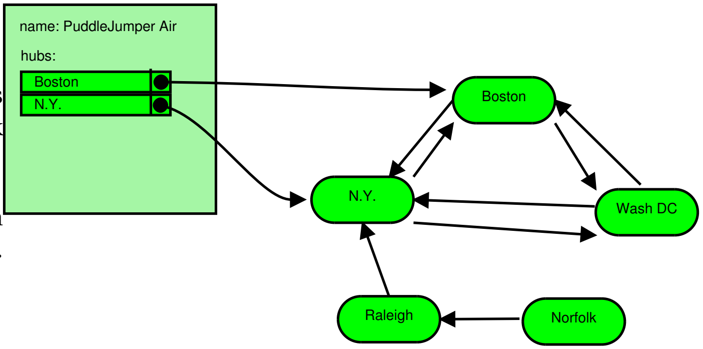


17.2 Garbage Collection

Garbage

Objects on the heap that can no longer be reached (in one or more hops) from any pointers in the activation stack (i.e., in local variables of active functions)or from any pointers in the static storage area (variables declared in C++ as "`static`")are called *garbage*.

Garbage Example



- In this example, if we assume that the airline object is actually a local variable in some function, then Norfolk and Raleigh appear to be garbage.
  - Unless there's some other pointer not shown in this picture, there's no way to get to either of them.
- Being garbage is not the same thing as "nothing points to it".

  - Raleigh is garbage even though something is pointing to it. Nonetheless, there is no way to get to Raleigh from the non-heap storage.

. . . .

Garbage Collection

Determining when something on the heap has become garbage is sufficiently difficult that many programming languages take over this job for the programmer.

The runtime support system for these languages provides *automatic garbage collection*, a service that determines when an object on the heap has become garbage anf automatically *scavenges* (reclaims the storage of) such objects.

. . . .

Java has GC

In Java, for example, although Java and C++ look very similar, there is no "delete" operator.

Java programmers use lots of pointers[1], many more than the typical C++ programmer.

But Java programmers never worry about deleting anything. They just trust in the garbage collector to come along eventually and clean up the mess.

. . . .

C++ Does Not

Automatic garbage collection really can simplify a programmer's life. Sadly, C++ does not support automatic garbage collection.

But how is this magic accomplished (and why doesn't C++ support it)? That's the subject of the remainder of this section.

. . . .

[1] Though, somewhat confusingly, they call them "references" instead of pointers. But they really are more like C++ pointers than like C++ references because you

- obtain one of these pointer/references by allocating an object on the heap via the operator `new`,
- can assign the value "null" to one of these to indicate that it isn't pointing at anything at all, and
- can assign a new address to one of these to make it point at some different object on the heap

All three of these properties are true of C++ pointers but not of C++ references. So Java "references" really are the equivalent of C++ "pointers", but by renaming them, Java advocates are able to boast that Java is a simpler language because it doesn't have pointers. That's more than a little disingenuous, IMO.

17.2.1 Reference Counting

Reference Counting

Reference counting is one of the simplest techniques for implementing garbage collection.

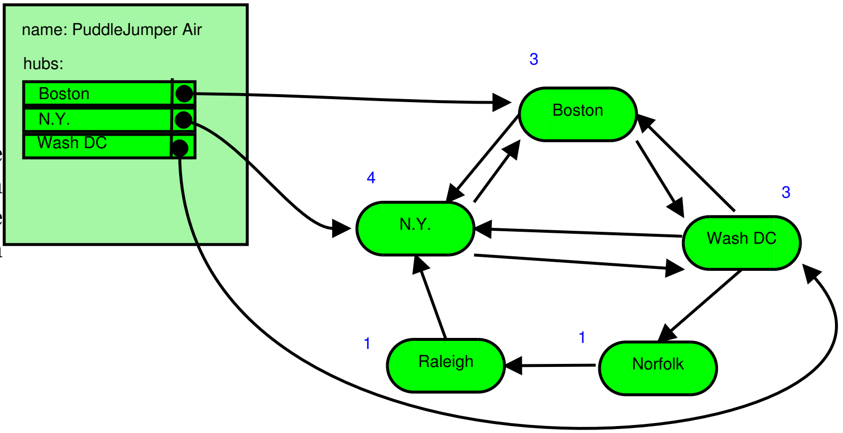
- Keep a hidden counter in each object on the heap. The counter will indicate how many pointers to that object exist.
  - Each time we reassign a pointer that used to point at this object, we decrement the counter.
  - Each time we reassign a pointer so that it now points at this object, we increment the counter.
- If that counter ever reaches 0, scavenge the object.

. . . .

Reference Counting Example

For example, here's our airline example with reference counts. Now, suppose that Wash DC loses its hub status.

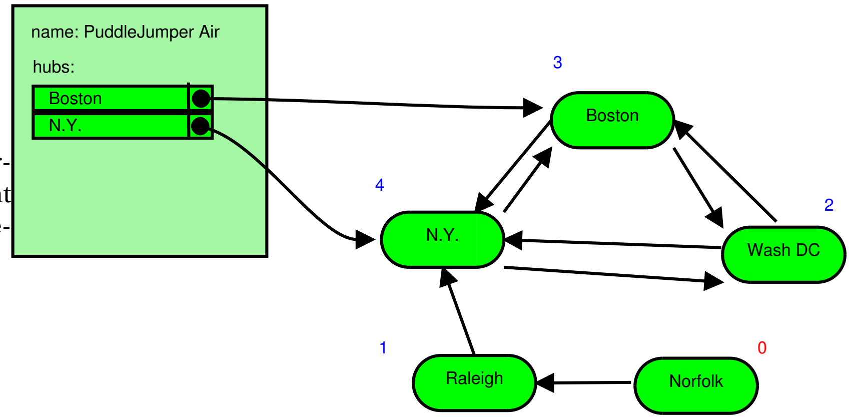
- The removal of the pointer from the airline object causes the reference count of Wash DC to decrease by one, but it's still more than zero, so we don't try to scavenge Wash DC.



. . . .

Reference Counting Example II

Now, suppose that Wash DC drops its service to Norfolk

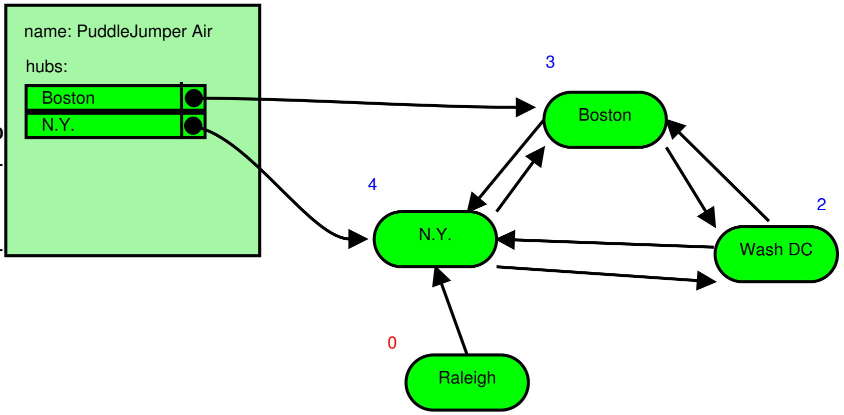


- When the pointer from Wash DC to Norfolk is removed, then the reference count of Norfolk decreases. In this case, it decreases to zero.

Reference Counting Example III

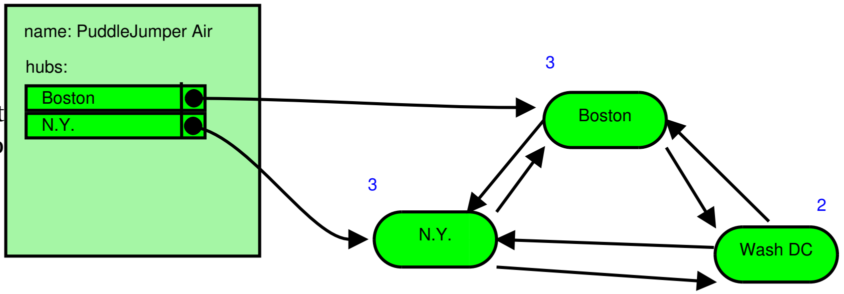
So the Norfolk object can be scavenged.

- When we do so, however, its pointer to Raleigh disappears, reducing Raleigh's reference count to zero.
  - That means that Raleigh can be scavenged.



Reference Counting Example IV

Doing that reduces N.Y.'s reference count, but the count stays above zero, so we don't try to scavenge N.Y.



Can we do this?

Implementing reference counting requires that we take control of pointers.

- To properly update reference counts, we would need to know whenever a pointer is assigned a new address (or null), whenever a pointer is created to point to a newly allocated object, and whenever a pointer is destroyed.

- Now, we can't do that for "real" pointers in C++.
  - But it is quite possible to create an ADT that looks and behaves much like a regular pointer.
  - And, by now, we know how to take control of assignment, copying, and destroying of ADT objects.

. . . .

A Reference Counted Pointer

Here is an (incomplete) sketch of a reference counted pointer ADT (which I will call a "smart pointer" for short).

```
template <class T>
class RefCountPointer {
  T* p;    ❶
  unsigned* count;

  void checkIfScavengable()  ❷
  {
    if (*count == 0)
      {
        delete count;
        delete p;
      }
  }

public:
  // This constructor is used to hand control of a newly
  // allocated object (*s) over to the reference count
  // system.  Example:
  //     RefCountPointer<PersonelRecord> p (new PersonelRecord());
  // It's critical that, once you create a reference counted
  // pointer to something, that you not continue playing with
  // regular pointers to the same object.
```

```
RefCountPointer (T* s)   ❸
  : p(s), count(new unsigned)
  {*count = 1;}

RefCountPointer (const RefCountPointer& rcp)
  : p(rcp.p), count(rcp.count)
  {++(*count);}   ❹

~RefCountPointer() {--(*count); checkIfScavengable();} ❺


RefCountPointer& operator= (const RefCountPointer& rcp)
  {
    ++(*(rcp.count));  ❻
    --(*count);
    checkIfScavengable();
    p = rcp.p;
    count = rcp.count;
    return *this;
  }

T& operator*() const {return *p;}  ❼
T* operator->() const {return p;}


bool operator== (const RefCountPointer<T>& ptr) const
{return ptr.p == p;}


bool operator!= (const RefCountPointer<T>& ptr) const
{return ptr.p != p;}
```

```
};
```

❶ The data stored for each of our smart pointers is p, the pointer to the real data, and count, a pointer to an integer counter for that data.

❷ This function will be called whenever we decrease the count. It checks to see if the count has reached zero and, if so, scavenges the object (and its counter).

❸ When the very first smart pointer is created for a newly allocated object, we set its counter to 1.

❹ When a smart pointer is copied, we increment its counter.

❺ When a smart pointer is destroyed, we decrement the count and see if we can scavenge the object.

❻ When a smart pointer is assigned a new value, we increment the counter of the object whose pointer is being copied, decrement the counter of the object whose pointer is being replaced, and check to see if that object can be scavenged.

❼ Other than that, the smart pointer has to behave like a real pointer, supporting the * and -> operators.

. . . .

Now, as I noted, this is an incomplete sketch. Right now, we have no means for dealing with null pointers, and we haven't provided the equivalent of a const pointer. Providing each of these would approximately double the amount of code to be shown (for a combined total of nearly four times what I've shown here). There's also some big issues when combining these smart pointers with inheritance and Object-Oriented programming.

But, hopefully, it's enough to serve as a proof of concept that this is really possible.

Is it worth the effort?

- Reference counting is fairly easy to implement.
  - Unlike the more advanced garbage collection techniques that we will look at shortly, it can be done in C++ because it does not require any special support from the compiler and the runtime system.

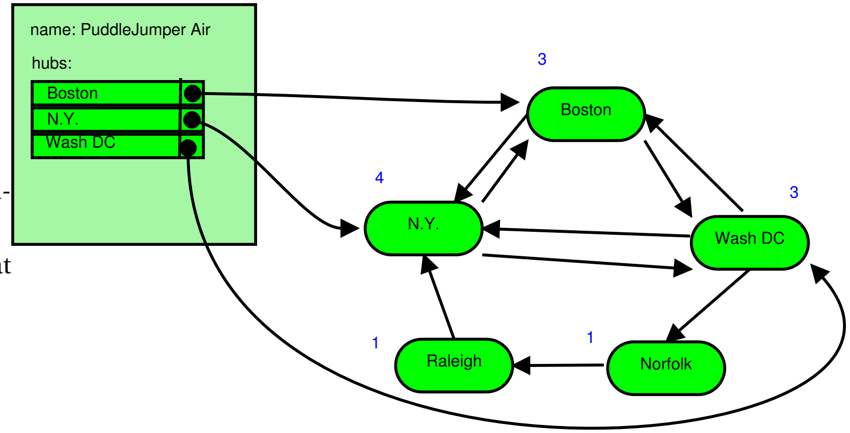
    There's a problem with reference counting, though.

- One that's serious enough to make it unworkable in many practical situations.

. . . .

Disappearing Airline

Let's return to our original airline example, with reference counts.

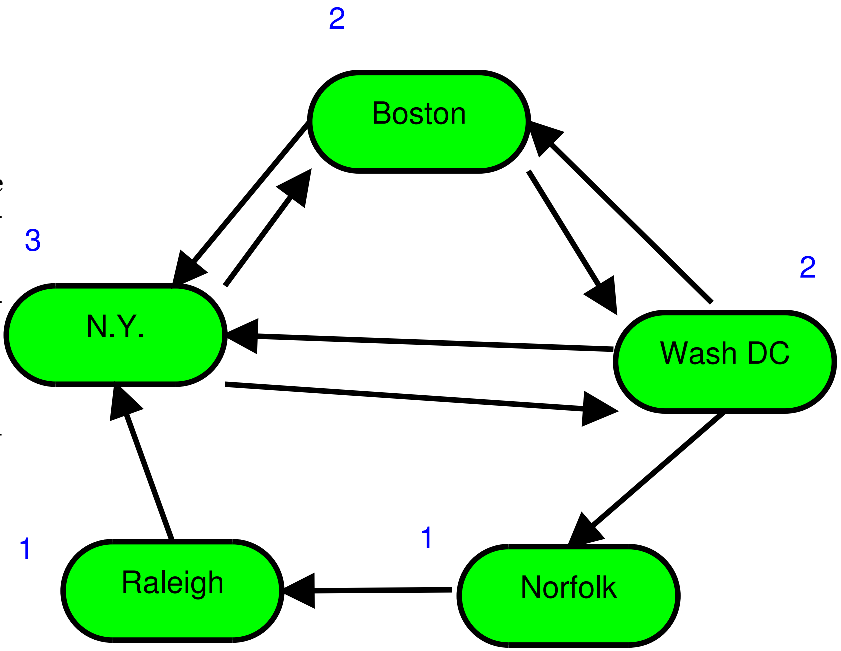


- Assume that
  - the airline object itself is a local variable in a function and that
  - we are about to return from that function.
- That object will therefore be destroyed, and its reference counted pointers to the three hubs will disappear.

. . . .

Leaky Airports

Here is the result, with the updated reference counts.



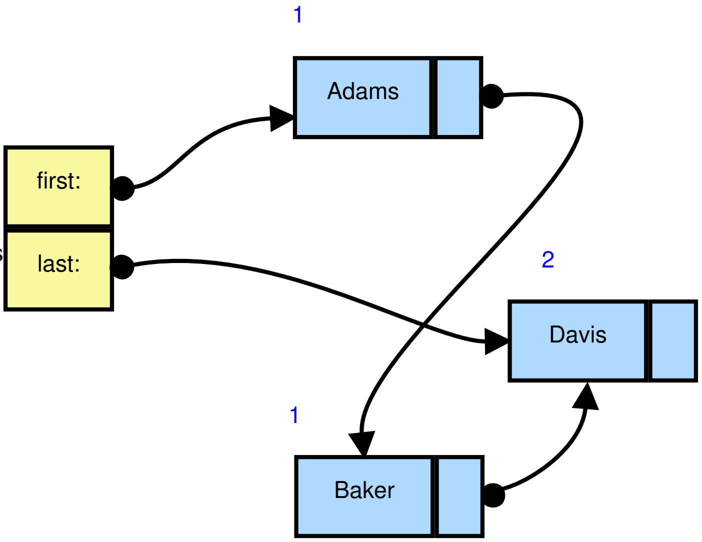
- Now, all of these airport objects are garbage, but none of them have zero reference counts.
  - Therefore none of them will be scavenged.
  - We're leaking memory, big time!

What went wrong? Let's look at our other examples.

Ref Counted SLL

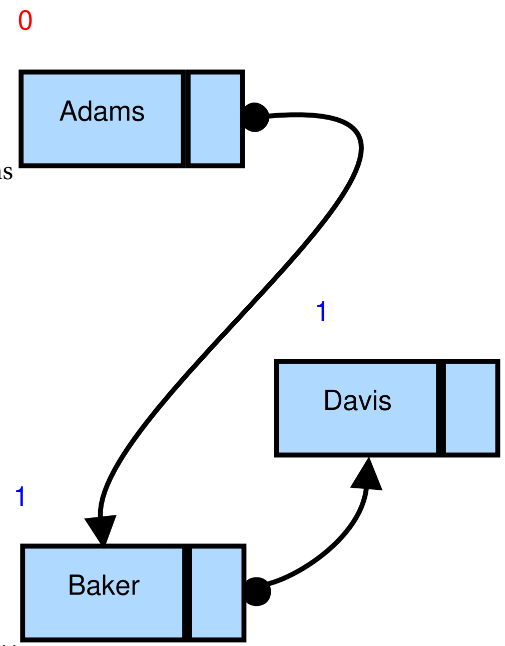
Here is our singly linked list with reference counts.

Assume that the list header itself is a local variable that is about to be destroyed.



. . . .

Ref Counted SLL II



- When the first and last pointers disappear, the reference count for Adams goes to zero.
  - So Adams can be scavenged.
- When it is, that will cause Baker's reference count to drop to zero.
  - When Baker is scavenged, Davis' count will drop to zero
- and it too will be scavenged.

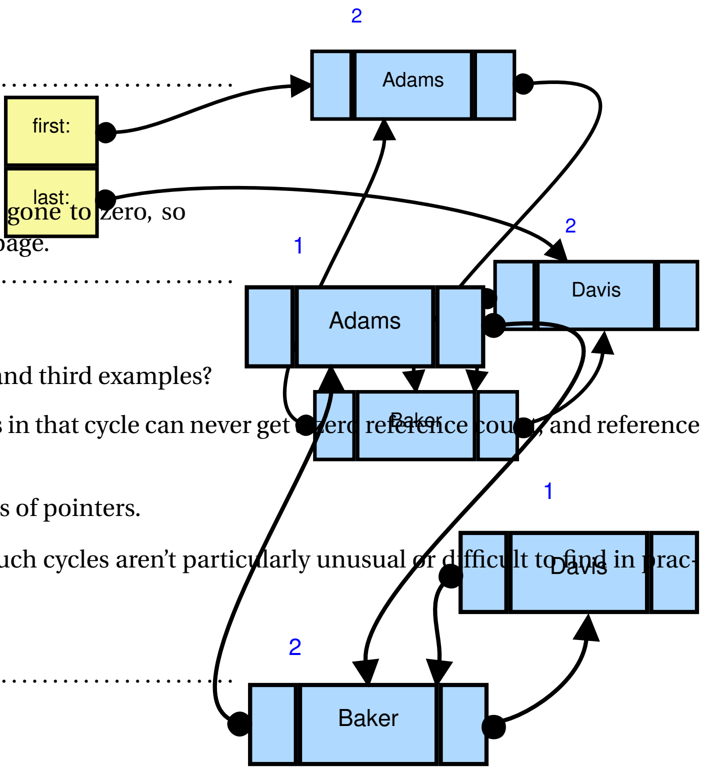
So that works just fine!

Ref Counted DLL

Now let's look at our doubly linked list.

Again, let's assume that the list header itself is a local variable that is about to be destroyed.

. . . .



Ref Counted DLL II

Here's the result.

Alas, we can see that none of the reference counters have gone to zero, so nothing will be scavenged, even though all three nodes are garbage.

. . . .

Reference Counting's Achilles Heel

What's the common factor between the failures in the first and third examples?

- It's the *cycles*. If the pointers form a cycle, then the objects in that cycle can never get zero reference count, and reference counting will fail.
- Reference counting won't work if our data can form cycles of pointers.
  - And, as the examples discussed here have shown, such cycles aren't particularly unusual or difficult to find in practical structures.

So a more general approach is needed.

. . . .

17.2.2 Mark and Sweep

Mark and Sweep

Mark and sweep is one of the earliest and best-known garbage collection algorithms.

- It works perfectly well with cycles, but
- requires some significant support from the compiler and run-time support system.

. . . .

Assumptions

The core assumptions of mark and sweep are:

- Each object on the heap has a hidden "mark" bit.
- We can find all pointers outside the heap (i.e., in the activation stack and static area)
- For each data object on the heap, we can find all pointers within that object.
- We can iterate over all objects on the heap

. . . .

The Mark and Sweep Algorithm

With those assumptions, the mark and sweep garbage collector is pretty simple:

```
void markAndSweep()
{
 // mark
 for (all pointers P on the run-time stack or
   in the static data area )
  {
    mark *P;
  }

 //sweep
 for (all objects *P on the heap)
   {
     if *P is not marked then
        delete P
     else
        unmark *P
   }
}

template <class T>
void mark(T* p)
{
  if *p is not already marked
    {
      mark *p;
      for (all pointers q inside *p)
        {
          mark *q;
        }
     }
}
```

The algorithm works in two stages.

- In the first stage, we start from every pointer outside the heap and recursively mark each object reachable via that pointer. (In graph terms, this is a depth-first traversal of objects on the heap.)
- In the second stage, we look at each item on the heap.
  - If it's marked, then we have demonstrated that it's possible to reach that object from a pointer outside the heap.
    - It isn't garbage, so we leave it alone (but clear the mark so we're ready to repeat the whole process at some time in the future).
  - If the object on the heap is not marked, then it's garbage and we scavenge it.

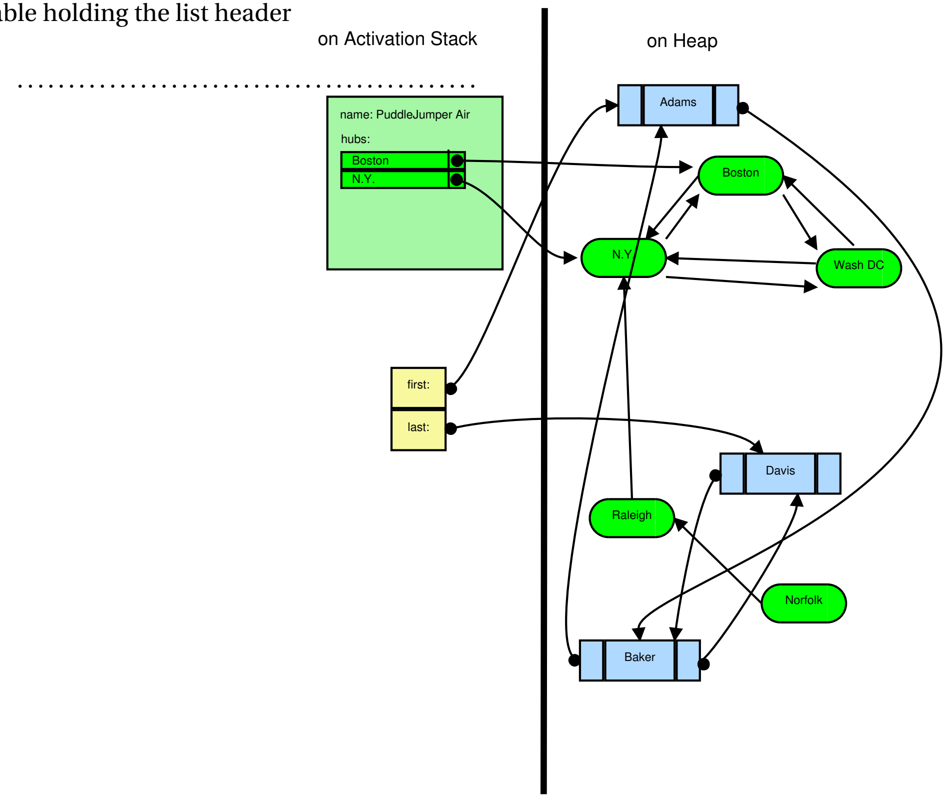
. . . .

Mark and Sweep Example

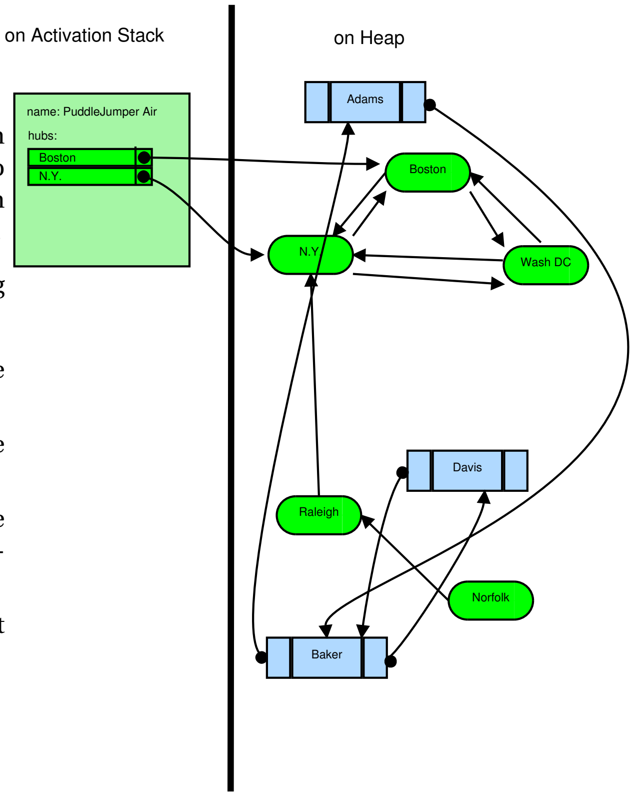
As an example, suppose that we start with this data.

Then, let's assume that the local variable holding the list header is destroyed.

Mark and Sweep Example II



- At some point later in time, the mark and sweep algorithm is started (typically in response to later code of ours trying to allocate something new in memory and the run-time system has discovered that we are running low on available storage.).
- The main algorithm begins the marking phase, looping through the pointers in the activation stack.
  - We have two. The first points to the Boston node. So we invoke the `mark()` function on the pointer to Boston.
    - The Boston node has not been marked yet, so we mark it.
  - Then the mark() function iterates over the pointers in the Boston object. It first looks at the N.Y. pointer and recursively invokes itself on that.
  - The N.Y. object has not been marked yet, so we mark it and then iterate over the pointers in N.Y.,



. . . .

We first come to the pointer to Boston, and recursively invoke mark() on that. But Boston is already marked, so we return immediately to the N.Y. call. Continuing on, we find a pointer to Wash DC. and invoke mark() on that.

The Wash DC object has not been marked yet, so we mark it and then iterate over the pointers in Wash DC. We first come to the pointer to Boston, and recursively invoke mark() on that. But Boston is already marked, so we return immediately to the N.Y. call. Again, that object is already marked so we immediately return to the earlier N.Y. call. That one has now visited all of its pointers, so it returns to the first Boston call.

The Boston call resumes iterating over its pointers, and finds a pointer to Wash DC. It calls mark() on that pointer, but Wash DC has already been marked, so we return immediately. The Boston call has now iterated over all of its pointers, so we return to the main mark and sweep algorithm.

That algorithm continues looking at pointers on the activation stack. We have a pointer to N.Y., and call mark() on that. But N.Y. is already marked, so we return immediately.

Mark and Sweep Example III

Once the mark phase of the main algorithm is complete,

- We have marked the Boston, N.Y., and Wash DC objects.
- The Norfolk, Raleigh, Adams, Baker, and Davis objects are unmarked.

. . . .

The Sweep Phase

In the sweep phrase, we visit each object on the heap.

- The three marked hubs will be kept, but their marks will be cleared in preparation for running the algorithm again at some time in the future.
- All of the other objects will be scavenged.

. . . .

Assessing Mark and Sweep

In practice, the recursive form of mark-and-sweep requires too much stack space.

- It can frequently result in recursive calls of the mark() function running thousands deep.
  - Since we call this algorithm precisely because we are running out of space, that’s not a good idea.

    Practical implementations of mark-and-sweep have countered this problem with an iterative version of the mark function that "reverses" the pointers it is exploring so that they leave a trace behind it of where to return to.

- Even with that improvement, systems that use mark and sweep are often criticized as slow. The fact is, tracing *every* object on the heap can be quite time-consuming. On virtual memory systems, it can result in an extraordinary number of page faults. The net effect is that mark-and-sweep systems often appear to freeze up for seconds to minutes at a time when the garbage collector is running. There are a couple of ways to improve performance.

. . . .

17.2.3 Generation-Based Collectors

Old versus New Garbage

In many programs, people have observed that object lifetime tends toward the extreme possibilities.

- temporary objects that are created, used, and become garbage almost immediately
- long-lived objects that do not become garbage until program termination

. . . .

Generational GC

Generational collectors take advantage of this behavior by dividing the heap into "generations".

- The area holding the older generation is scanned only occasionally.
- The area holding the youngest generation is scanned frequently for possible garbage.
  - an object in the young generation area that survives a few garbage collection passes is moved to the older generation area

. . . .

The actual scanning process is a modified mark and sweep. But because relatively few objects are scanned on each pass, the passes are short and the overall cost of GC is low.

To keep the cost of a pass low, we need to avoid scanning the old objects on the heap. The problem is that some of those objects may have pointers to the newer ones. Most generational schemes use traps in the virtual memory system to detect pointers from "old" pages to "new" ones to avoid having to explicitly scan the old area on each pass.

17.2.4 Incremental Collection

Incremental GC

Another way to avoid the appearance that garbage collection is locking up the system is to modify the algorithm so that it can be run one small piece at a time.

- Conceptually, every time a program tries to allocate a new object, we run just a few mark steps or a few sweep steps,
- By dividing the effort into small pieces, we give the illusion that garbage collection is without a major cost.

  There is a difficuty here, though. Because the program might be modifying the heap while we are marking objects, we have to take extra care to be sure that we don't improperly flag something as garbage just because all the pointers to it have suddenly been moved into some other data structure that we had already swept.
- In languages like Java where parallel processes/threads are built in to the language capabilities, systems can take the incremental approach event further by running the garbage collector in parallel with the main calculation.

  Again, special care has to be taken so that the continuously running garbage collector and the main calculation don't interfere with one another.

. . . .

17.3 Strong and Weak Pointers

Doing Without

OK, garbage collection is great if you can get it.

- But C++ does not provide it, and C++ compilers don't really provide the kind of support necessary to implement mark ans sweep or the even more advanced forms of GC.
- So what can we, as C++ programmers do, when faced with data structures that need to share heap data with one another?

. . . .

Ownership

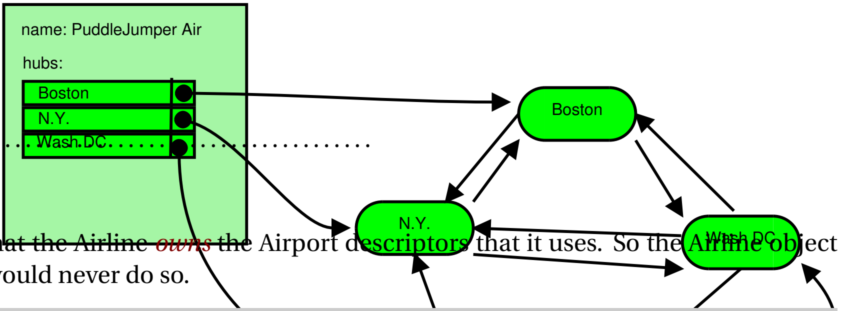
One approach that works in many cases is to try to identify which ADTs are the *owners* of the shared data, and which ones merely use the data.

- The owner of a collection of shared data has the responsibility for creating it, sharing out pointers to it, and deleting it.
- Other ADTs that share the data without owning it should never create or delete new instances of that data.

. . . .

Ownership Example

In this example that we looked at earlier, we saw that if both the Airline object on the left and the Airport objects on the right deleted their own pointers when destroyed, our program would crash.



. . . .

Ownership Example

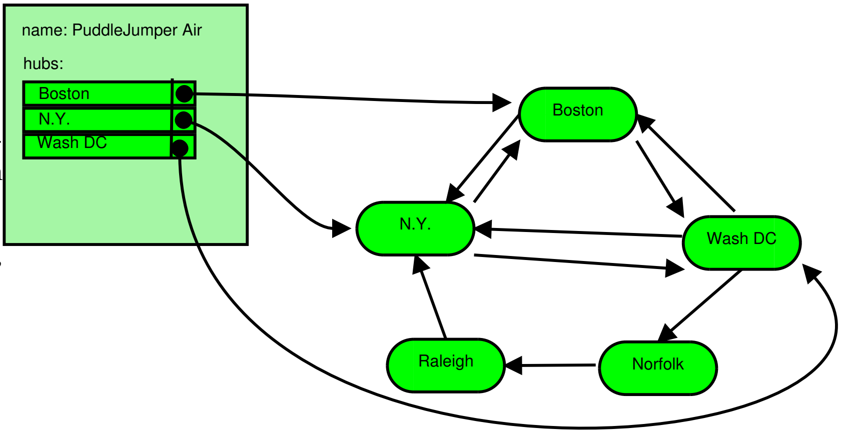
We could improve this situation by deciding that the Airline *owns* the Airport descriptors that it uses. So the Airline object would delete the pointers it has, but the Airports would never do so.

```
class Airport
{
   ⋮

private:
   vector<Airport*> hasFlightsTo;
};

Airport::~Airport()
{
  /* for (int i = 0; i < hasFlightsTo.size(); ++i)
```

```
      delete hasFlightsTo[i]; */
}

class AirLine {
   ⋮
   string name;
   map<string, Airport*> hubs;
};


AirLine::~Airline()
{
   for (map<string, Airport*>::iterator i = hubs.begin;
        i != hubs.end(); ++i)
     delete i->second;
}
```

. . . .

Ownership Example



Thus, when the airline object on the left is destroyed, it will delete the Boston, N.Y., and Wash DC objects.

- Each of those will be deleted exactly once, so our program should no longer crash.

- This solution isn't perfect. The Norfolk and Raleigh objects are never reclaimed, so we do wind up leaking memory. The problem is that, having decided that the Airline owns the Airport descriptors, we have some Airport objects with no owner at all.

. . . .

Asserting Ownership

I would probably resolve this by modifying the Airline class to keep better track of its Airports.

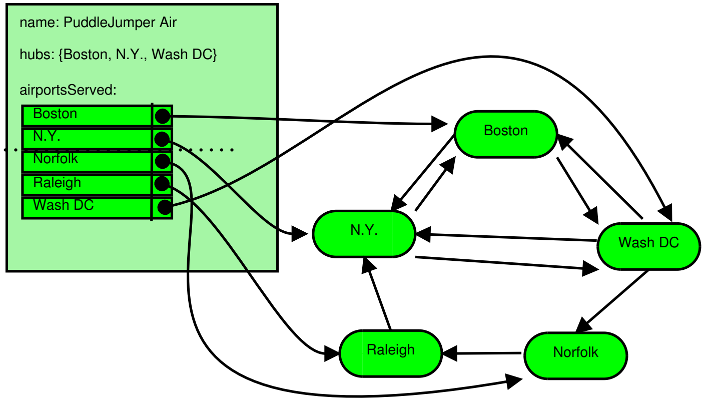
```
class AirLine {
   ⋮
   string name;
   set<string> hubs;
   map<string, Airport*> airportsServed;
};
```

```
AirLine::~Airline()
{
   for (map<string, Airport*>::iterator i = airportsServed.begin;
        i != airportsServed.end(); ++i)
     delete i->second;
}
```

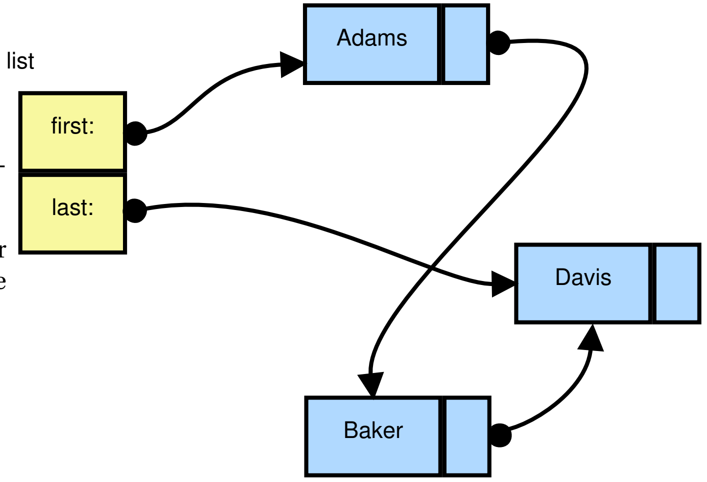
Asserting Ownership (cont.)

The new map tracks all of the airports served by this airline, and we use a separate data structure to indicate which of those airports are hubs.

Now, when an airline object is destroyed, all of its airport descriptors will be reclaimed as well.



Ownership Can Be Too Strong



Ownership is sometimes a bit too strong a relation to be useful.

- In this example, if we simply say that the list header owns the nodes it points to, then we would delete the first and last node and would leave Baker on the heap.

- And if we say that the nodes owned the other nodes that they point to *and* that the list header owns the ones it points to, we would delete the last node twice.

. . . .

Strong and Weak Pointers

We can generalize the notion of ownership by characterizing the various pointer data members as strong or weak.

- A *strong pointer* is a pointer data member that indicates that the object pointed to must remain in memory.

- A *weak pointer* is a pointer data member that is allowed to point to data that might have been deleted.

  – (Obviously, we never want to follow a weak pointer unless we are sure that the data has not, in fact, been deleted.)

When an object containing pointer data members is destroyed, it deletes its strong pointer members and leaves its weak ones alone.

. . . .

Strong and Weak SLL

In this example, if we characterize the pointers as shown:

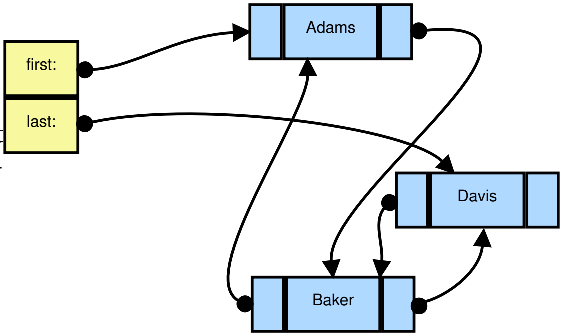
```
struct SLNode {
   string data;
   SLNode* next; // strong
     ⋮
   ~SLNode () {delete next;}
};

class List {
   SLNode* first; // strong
   SLNode* last;  // weak
public:
     ⋮
   ~List()
    {
      delete first;  // OK, because this is strong
      /*delete last;*/ // Don't delete. last is weak.
     }
};
```

then our program will run correctly.

. . . .

Picking the Strong Ones

The key idea is to select the smallest set of pointer data members that would connect together all of the allocated objects, while giving you exactly one path to each such object.



. . . .

Strong and Weak DLL

Similarly, in a doubly linked list, we can designate the pointers as follows:

```
struct DLNode {
  string data;
  DLNode* prev; // weak
  DLNode* next; // strong
    ⋮
  ~DLNode () {delete next;}
};

class List {
  DLNode* first; // strong
  DLNode* last;  // weak
public:
    ⋮
  ~List() {delete first;}
};
```

and so achieve a program that recovers all garbage without deleting anything twice.

. . . .

17.4 Coming Soon to C++: std Reference Counting

C++11 Adds Reference Counting

The new C++11 standard (endorced by ISO in Sept 2011) contains proposals for smart pointer templates, quite similar in concept to the *RefCountPointer* discussed earlier.

- Some compilers may already offer versions.

. . . .

shared and weak ptrs

There are two primary class templates involved

- *shared_ptr<T>* provides a strong pointer with an associated reference counter.
  - Shared pointers may be freely assigned to one another.
  - If all shared pointers to an object on the heap are destroyed or are reassigned to point elsewhere, then that reference count will drop to zero and the object on the heap will be deleted.
- *weak_ptr<T>* provides a weak pointer to an object that has an associated reference counter, but
  - creating, copying, assigning, or destroying weak pointers does not affect the reference counter value.
- Strong and weak pointers may be copied to one another.

. . . .

We can illustrate these with our doubly linked list. (I would not do this in real practice, because the prior version with informally identified strong and weak pointers is simpler and probably slightly faster.)

```
struct DLNode {
   string data;
   weak_ptr<DLNode> prev;
   shared_ptr<DLNode> next;
     ⋮
   ~DLNode ()
     {
       // do nothing - the reference counting will take care of it
      }
};

class List {
   shared_ptr<DLNode> first;
   weak_ptr<DLNode> last;
public:
     ⋮
   ~List() {/* do nothing */}
};
```

Once this is set up, we use those four data members largely as if they were regular pointers. The only difference is when constructing new smart pointers. Some sample code:

```
void addToFront (List& list, string newData)
{
   shared_ptr<DLNode> newNode (new DLNode()); ❶
   newNode->data = newData;             ❷
   if (first == shared_ptr<DLNode>()) ❸
     {
      // list is empty
      first = last = newNode;    ❹
     }
   else
     {
      newNode->next = first;
      first->prev = newNode;
      first = newNode;
     }
}
```

❶ This line allocates a new object and immediately hands it off to the reference counting system by creating a shared pointer to it (and losing the "real" pointer so that we aren't even tempted to play around with it directly).

❷ Looking at this line, there's nothing to tell us that we're working with a smart pointer instead of a regular pointer.

❸ The default constructor for shared and weak pointers presents the equivalent of a null pointer.

❹ Notice that, in this statement, we assign a shared pointer to a weak pointer, then assign that weak pointer to a different shared pointer.

Once you figure out which data members should be strong (shared) and which should be weak, you can pretty much forget the difference afterwards.

17.5 Java Programmers Have it Easy

Java Programmers Have it Easy

Java has included automatic garbage collection since its beginning.

- From a practical point of view, sharing in Java is actually easier (and more common) than deep copying.
- Java programmers typically are unconcerned with many of the memory management errors that C++ programmers must strive to avoid.

. . . .

C++ programmers may sometimes sneer at the slowdown caused by garbage collection. The collector implementations, however, continue to evolve. In fact, current versions of Java commonly offer multiple garbage collectors, one which can be selected at run-time in an attempt to find one whose run-time characteristics (i.e., how aggressively it tries to collect garbage and how much of the time it can block the main program threads while it is working) that matches your program's needs.

Java programmers sometimes face an issue of running out of memory because they have inadvertently kept pointers to data that they no longer need. This is a particular problem in implementing algorithms that use caches or memoization to keep the answers to prior computations in case the same result is needed again in the future. Because of this, Java added a concept of a weak reference (pointer) that can be ignored when checking to see if an object is garbage and that gets set to null if the object it points to gets collected.

Part V

OOA&D: Use Cases

Chapter 18

Use cases

18.1 Scenarios

Scenarios

Most OOAD is scenario-driven.

- Scenarios are "stories" that describe interactions between a system and its users.
- Usually in informal natural language
  - Often with many variants on a common theme to accommodate errors, exceptional cases, alternative processing, etc.
- Gathered (during Elaboration) from documents and interviews with domain experts

. . . .

Example: Assessment Scenarios

1. An instructor prepares three new questions for an upcoming test. One is an essay question, one is a multiple-choice question, and the third calls for students to perform a physical experiment to submit tabulated data from that experiment. The instructor proceeds to compose a test from these three questions plus an existing question from his personal store.
2. A student wishes to attempt a test. She approaches the instructor who provides a copy of the test on which the student is supposed to write her answers.. The instructor authorizes the student to begin the assessment session.
3. Same as #2, but the student is given a copy of the questions and a separate form on which to record her responses.
4. Same as #3, but the student brings a set of blue books that are approved by the instructor as a separate form on which to record her responses.
5. A student wishes to attempt a test. He approaches the instructor, but the instructor will not authorize the assessment session because the test is not available until next day.
6. A student begins an assessment session. For each question on the test, the student reads the question and takes note of the expected response types. The first question is an essay question. The student prepares a few paragraphs of text as her response. The second question is a multiple-choice question. The student selects one choice as her response. The third question calls for the student to conduct an activity outside of the current session, so the student suspends the assessment session, with the intention of resuming it after the external activity has been completed.
7. An instructor begins to process the responses of a collection of students. All responses are for the same assessment. For each question in that assessment, the instructor obtains the question's rubric. Then for each response, the instructor obtains the response for that question, applies the rubric, and records the score as part of the assessment result. Once all questions have been scored, the instructor computes and records the assessment score in the assessment result.
8. An instructor is processing an question response to an essay question. The rubric consists of a series of concepts that could be considered valid, coupled with a score value for each. The instructor reads the response, keeping a running total of the value of all rubric concepts recognized. That total is compared to the question's normal minimum and normal maximum and adjusted, if necessary, to fall within those bounds. The adjusted total is recorded as the question score for that question response.

   If the question score is less than the normal max, then the instructor adds a list of the rubric concepts not contained in the response as feedback within the question outcome.

. . . .

18.2 Use Cases

Use Cases

A *use case* is

- a collection of scenarios
- related to a common user goal
- expressed in step-by-step form

. . . .

Where do use cases come from?

Typically these arise from analysis of documents provided to the analysis team or from interviews conducted with the domain experts. (This might be a good time to review the different stages of analysis and modeling that we have discussed.)

. . . .

What are use cases good for?

What do we do with use-cases once we have them?

- Use them to guide analysis and design
  - Validate models by walking through use cases to see if model supports the scenario
- Sometimes used as requirements statements

. . . .

18.2.1 Evolving a Collection of Use cases

Evolving a Collection of Use cases

1. Start with a collection of scenarios

2. Group by common user goals
3. Define the Actors
4. Give brief descriptions of use cases
5. Give a detailed description of the basic path
6. Add alternative paths

. . . .

Start with a collection of scenarios
The informal scenarios are your starting point.

. . . .

Example 1. Assessment Scenarios

- Consultation with domain experts has revealed a standard set of terminology (QTI) related to assessments, particularly automated assessments. Accordingly we replace certain informal terms by their more domain-appropriate equivalents ("question" ⇒ "item", "student" ⇒ "candidate") and the role of "instructor" is split into distinct classes of "author", "proctor", and "scorer".

1. An author prepares three new items for an upcoming test. One is an essay question, one is a multiple-choice question, and the third calls for students to perform a physical experiment to submit tabulated data from that experiment. These items become part of that author's personal item bank. The author proceeds to compose a test from these three items plus an existing item (a fill-in-the-blank question) from the item bank.
2. A candidate wishes to attempt a test. She approaches the proctor who provides a copy of the test on which the candidate is supposed to write her answers.. The proctor authorizes the candidate to begin the assessment session.
3. Same as #2, but the candidate is given a copy of the questions and a separate form on which to record her responses.

4. Same as #3, but the candidate brings a set of blue books that are approved by the proctor as a separate form on which to record her responses.

5. A candidate wishes to attempt a test. He approaches the proctor, but the proctor will not authorize the assessment session because the test is not available until next day.

6. A candidate attempts a test. She approaches the proctor who provides a copy of the test and a (possibly separate) empty response form. The proctor authorizes the student to begin the assessment session. For each item on the test, the candidate reads the item body and takes note of the item variables. The first item is an essay question. The candidate prepares a few paragraphs of text as her item response. The second item is a multiple-choice question. The student selects one choice as her item response. The third question calls for the candidate to conduct an activity outside of the current session, so the student suspends the assessment session, with the intention of resuming it after the external activity has been completed.

7. A scorer begins to process the responses of a collection of candidates. All responses are for the same assessment. For each item in that assessment, the scorer obtains the item's rubric. Then for each response, the scorer obtains the item response for that item, applies the rubric, and records the item outcome as part of the assessment result. Once all items have been scored, the scorer computes and records the assessment outcome in the assessment result.

8. A scorer is processing an item response to an essay question. The rubric consists of a series of concepts that could be considered valid, coupled with a score value for each. The scorer reads the response, keeping a running total of the value of all rubric concepts recognized. That total is compared to the item's normal minimum and normal maximum and adjusted, if necessary, to fall within those bounds. The adjusted total is recorded as the item outcome for that item response.

   If the item outcome is less than the normal max, then the scorer adds a list of the rubric concepts not contained in the response as feedback within the item outcome.

- *CRC cards and class relationship diagrams should also be updated accordingly.*

Group by common user goals

A use case describes a collection of scenarios related by common use goals.

- Note that a given scenario might lack a clear goal or might span multiple goals.

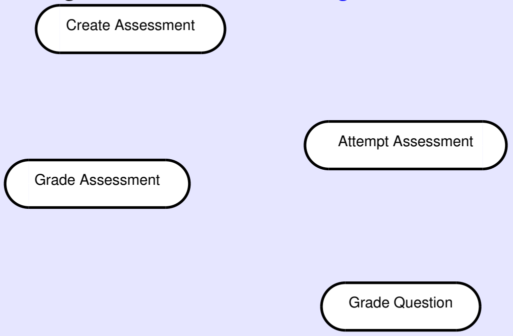
. . . .

Group by common user goals

Example 2. Assessment User Goals

- Create an assessment
- Attempting an assessment
- Grading an assessment
- Grading a question

We can summarize these with the initial stage of a UML *Use case Diagram*:



. . . .

Define the Actors

An *actor* (in this context) is a role that a user plays in this system.
An actor is usually an entity that behaves spontaneously to initiate an interaction.

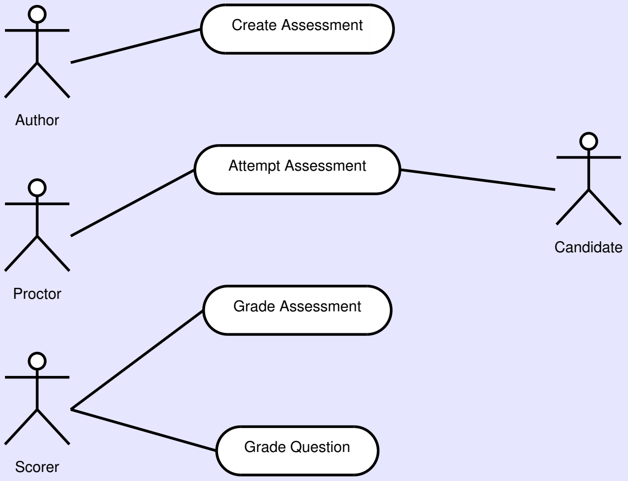
- Often include the major users and stakeholders in the system
- Relating use cases to actors helps identify the domain experts who need to be consulted on each case to obtain details, validate for accuracy, etc.

  Later, these relations help characterize user interfaces.

. . . .

Defining the Actors

Example 3. Assessment Actors



The connections between actors and use cases indicate that the actor participates in the use case.

. . . .

Give brief descriptions of use cases

For each of the use cases, describe briefly what goes on in this case.

You can pull heavily on the original text of the scenarios, but generalize to pull in the identified actors. You may abstract some details.

....

Example 4. Assessment UC Brief Descriptions

Create Assessment An author creates a new assessment, using a variety of different types of items (questions). Some of these items are created specifically for this assessment. Others are pulled from the author's item bank.

Attempt Assessment A candidate attempts an assessment. The proctor provides a copy of the assessment and a (possibly separate) empty response document. The proctor authorizes the student to begin the assessment session and can end the session when the candidate is finished or upon expiration of a time limit. During the session, the candidate prepares responses to items in the assessment. At the end of the session, the student returns the response document to the proctor.

Grade Assessment A scorer begins to process the response documents of a collection of candidates. All response documents are for the same assessment. The scorer may opt to process responses on an item by item basis (grading all student responses to a single item) or on a candidate by candidate basis (grading all item responses for a given candidate). Once all items have been scored, the scorer computes and records the assessment outcome in the assessment result.

Grade Item A scorer begins to grade an item response within a response document to an assessment. The scorer obtains the item's rubric for that assessment, applies that rubric to the item response, and records the resulting score and, optionally, feedback as part of the assessment result.

Give a detailed description of the basic path

The basic path through your use case is the normal, everything-is-going-right, processing.

- This is expressed as a numbered sequence of steps.
- Eventually you want enough detail to model this as a set of messages exchanged between objects, but you may need multiple iterations to get there.

. . . .

Example 5. Assessment UC Basic Paths

Create Assessment

1. Author creates items of varying kinds, adding each to his/her item bank.
2. Author creates an empty assessment, adding a title and indicating the grading policy (graded immediately or in batch, grades recorded in grade book or not), policy on re-attempts permitted, dates & time of availability, and security policy (suspend & resume permitted, open/closed book, etc.).
3. Author selects questions from item bank and adds them to the assessment, assigning point value as he/she does so.
4. Author concludes creation of the assessment, making it available to the appropriate proctors.

Attempt Assessment

1. A candidate indicates to a proctor their desire to take an assessment. (This may involve simply showing up at a previously-schedule time and place when the assessment is going to be available.)
2. The proctor provides the candidate with a copy of the assessment and an empty response document.
3. The proctor authorizes the start of the assessment session.
4. The candidate reads the assessment, and repeatedly selects items from it. For each selected item, the candidate creates a response, adding it to the response document.
5. The proctor ends the assessment session after time has expired.
6. The candidate returns the response document to the proctor.

Grade Assessment

1. The scorer begins with an assessment and a collection of response documents.

2. For each item in the assessment, the scorer obtains the item's rubric. Then for each response document, the scorer goes to the item response for that same item, grades the response using that rubric, and adds the resulting core and (if provided) feedback to the result document for that response document.
3. When all items have been graded, then the scorer computes a total score for each results document.
4. The scorer add the score from each result document to the grade book.

Grade Item

Given an assessment item, its rubric, and an item response,

1. The scorer applies the rubric to the item response to obtain a raw score and, possibly, feedback text for the item.
2. The raw score is scaled to match the point value in the rubric to the point value of the item in the assessment. If need be, the scaled score is subjected to the minimum/maximum point constraints on the item.
3. The results is returned as the result score, together with any feedback.

Add alternative paths

A use case is a *collection* of scenarios. The basic path constitutes a single scenario.

- The basic path constitutes a single scenario.
  - not necessarily one of the original scenarios
- Additional scenarios are introduced as alternative paths

. . . .

Example 6. Assessment UC Alternate Paths

Create Assessment

1. Author creates items of varying kinds, adding each to his/her item bank.
2. Author creates an empty assessment, adding a title and indicating the grading policy (graded immediately or in batch, grades recorded in grade book or not), policy on re-attempts permitted, dates & time of availability, and security policy (suspend & resume permitted, open/closed book, etc.).
3. Author selects questions from item bank and adds them to the assessment, assigning point value as he/she does so.
4. Author concludes creation of the assessment, making it available to the appropriate proctors.

Alternative: *Edit assessment*

1. Author edits an existing assessment, possibly altering one or more of the assessment properties.

Alternative: *Save assessment*

1. Author concludes this work session, saving the assessment for later editing without releasing it to the proctors.

Attempt Assessment

1. A candidate indicates to a proctor their desire to take an assessment. (This may involve simply showing up at a previously-schedule time and place when the assessment is going to be available.)
2. The proctor provides the candidate with a copy of the assessment and an empty response document.
3. The proctor authorizes the start of the assessment session.
4. The candidate reads the assessment, and repeatedly selects items from it. For each selected item, the candidate creates a response, adding it to the response document.
5. The proctor ends the assessment session after time has expired.
6. The candidate returns the response document to the proctor.

18.3 Organizing Use Cases

Organizing Use Cases

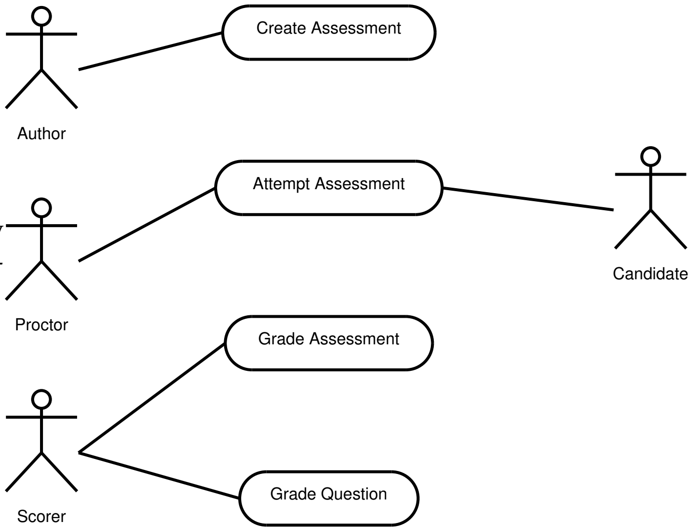
- A 10 person-year project may typically have a dozen use cases (Fowler) with lots of alternative paths.
- In some cases, it's preferred to break some of the alternative paths to have simpler, related use cases.
  - UML recognizes 3 relations between use cases.
- The collection of use cases, actors, and the relations among them constitutes our *use case model*.

. . . .

18.3.1 Includes

One Use Case Invoking Another

In our current model, the steps of *Grade Question* are actually a description of something that happens inside *Grade Assessment*.



. . . .

Example

Grade Assessment

1. The scorer begins with an assessment and a collection of response documents.
2. For each item in the assessment, the scorer obtains the item's rubric. Then for each response document, the scorer goes to the item response for that same item, grades the response using that rubric, and adds the resulting score and (if provided) feedback to the result document for that response document.
3. When all items have been graded, then the scorer computes a total score for each results document.
4. The scorer add the score from each result document to the grade book.

Alternative: *Candidate by Candidate Scoring*

1. For each candidate, the scorer goes through each of the items. For each item, the scorer obtains the item's rubric, grades the item response using that rubric, adds the resulting score and (if provided) feedback to the result document for that response document.

Grade Item
Given an assessment item, its rubric, and an item response,

1. The scorer applies the rubric to the item response to obtain a raw score and, possibly, feedback text for the item.
2. The raw score is scaled to match the point value in the rubric to the point value of the item in the assessment. If need be, the scaled score is subjected to the minimum/maximum point constraints on the item.
3. The results is returned as the result score, together with any feedback.

Alternative:*Essay Question Scoring*

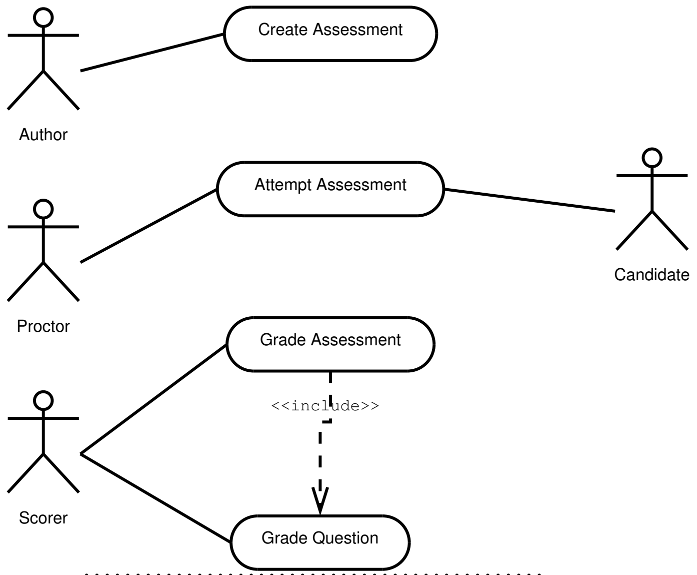
1. The item is an essay question.

   The rubric consists of a series of concepts that could be considered valid, coupled with a score value for each.

   (a) The scorer reads the response, keeping a running total of the scores of all rubric concepts recognized.

The Includes Relation

This is an example of an *includes* relation between use cases.



. . . .

18.3.2 Generalization

The Generalization Relationship

The *generalization* relation between use cases indicates that one use case meets the same user goal as the other but overrides one or more steps of the procedure for meeting that goal.

- For example, we will shortly look at sequence diagrams as a way of mapping our analysis & design models onto use cases.
  - If would be hard to document the two paths in the following use case in that fashion:

. . . .

Example

Grade Assessment

1. The scorer begins with an assessment and a collection of response documents.
2. For each item in the assessment, the scorer obtains the item's rubric. Then for each response document, the scorer goes to the item response for that same item, grades the response using that rubric, and adds the resulting score and (if provided) feedback to the result document for that response document.
3. When all items have been graded, then the scorer computes a total score for each results document.
4. The scorer add the score from each result document to the grade book.

Alternative: *Candidate by Candidate Scoring*

1. For each candidate, the scorer goes through each of the items. For each item, the scorer obtains the item's rubric, grades the item response using that rubric, adds the resulting score and (if provided) feedback to the result document for that response document.

Splitting a Use Case

So we might choose to break this into two distinct cases:

. . . .

Example

Grade Assessment

1. The scorer begins with an assessment and a collection of response documents.
2. For each item in the assessment, the scorer obtains the item's rubric. Then for each response document, the scorer goes to the item response for that same item, grades the response using that rubric, and adds the resulting score and (if provided) feedback to the result document for that response document.

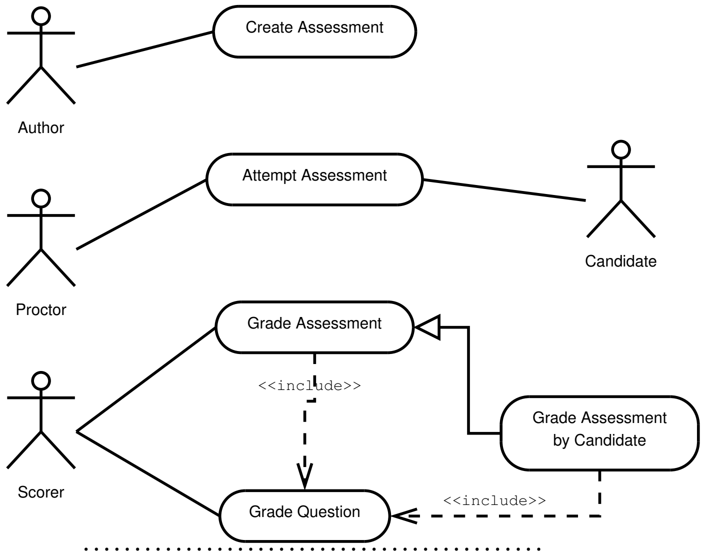
3. When all items have been graded, then the scorer computes a total score for each results document.
4. The scorer add the score from each result document to the grade book.

Grade Assessment by Candidate

1. The scorer begins with an assessment and a collection of response documents.
2. For each candidate, the scorer goes through each of the items. For each item, the scorer obtains the item's rubric, grades the item response using that rubric, adds the resulting score and (if provided) feedback to the result document for that response document.
3. When all items have been graded, then the scorer computes a total score for each results document.
4. The scorer adds the score from each result document to the grade book.

Generalization

The relation between these two use cases is shown like this:



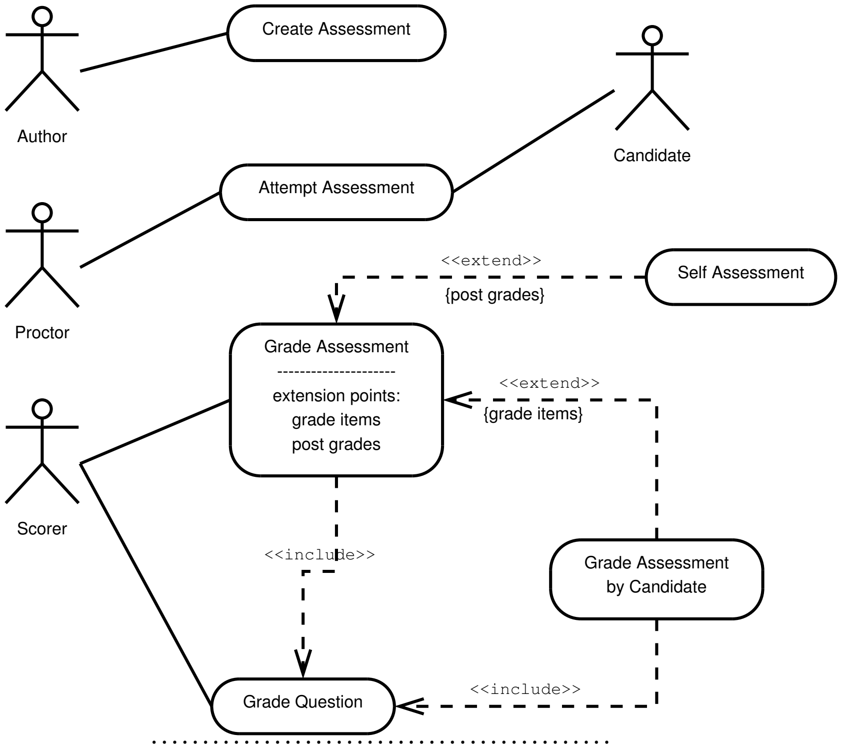
18.3.3 Extends

Extends

A more "disciplined" version of generalization.

- The base use case specifies specific *extension points* at which other use cases may override steps.
- The derived use cases announce which extension points they are going to override.

Extends



18.4 How to Use a Use Case

How to Use a Use Case

- As an aid to analysis: read through the case, looking for objects and interactions not listed in the current model.

- As an aid to validation: Model the use case as a series of passed messages, recording the interactions in a *sequence diagram*. Update the model as necessary so that a cause-and-effect chain can be demonstrated from the start of the use case to its end.
- As a statement of requirements: some companies now use use-cases in place of older forms of requirements statements

. . . .

Chapter 19

Interaction Diagrams

Interaction Diagrams

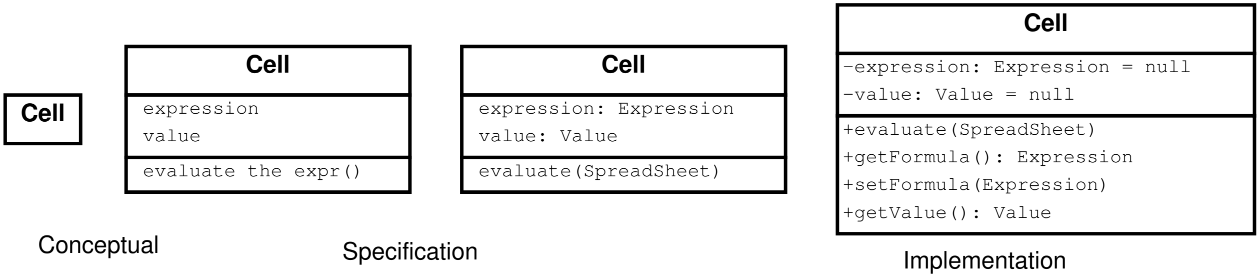
UML Interaction diagrams describe how groups of objects collaborate in some behavior.

Important: these diagrams are about *objects*. The UML class diagrams that we looked at earlier are about (surprise!) *classes*.

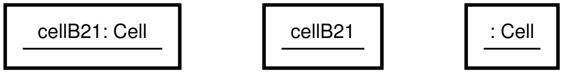
. . . .

UML: Objects vs. Classes

Remember that, in UML, these represent classes:



But now we want objects:



- Note the underlining and non-bold face.
- The syntax for the text describing the object is similar to the attributes in the class diagrams.

. . . .

Two Kinds of Interaction Diagrams

We'll use these objects in two different kinds of interaction diagrams:

- Sequence diagrams
- Collaboration diagrams

. . . .

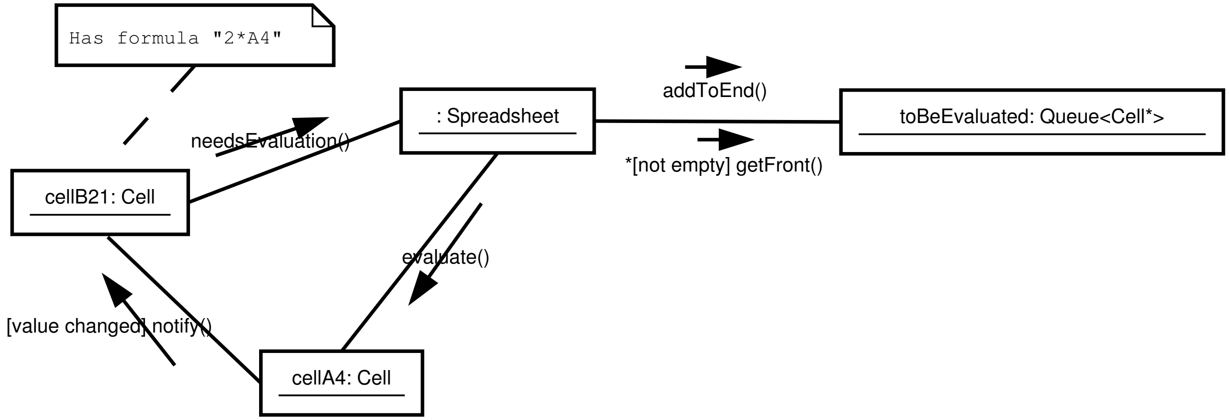
19.1 Collaboration Diagrams

Collaboration Diagrams

An older technique for diagramming collaborations. These are easier to read for small, simple cases, but less useful for complicated sequences of messages.

. . . .

Components of a Collaboration Diagram



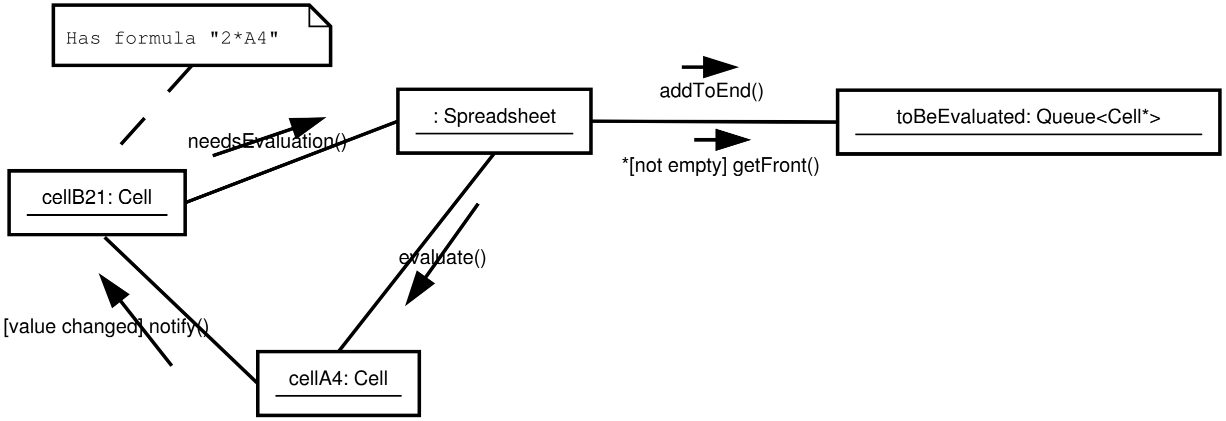
- The main components of a collaboration diagram are *objects*
- Objects that exchange messages are joined by *links*, shown as solid lines running from one object to another.
- If one object passes a message to (calls a member function of) another, we write the message name alongside the link, with an arrow showing the direction of the message.

....

19.1.1 Control Flow

Control Flow: guards

There is a limited capacity for indicating control flow.



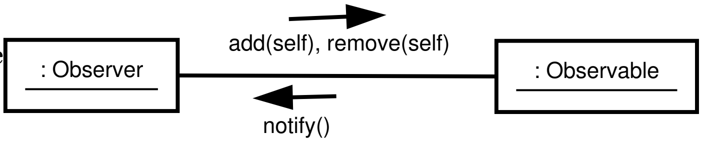
- A *guard* indicates that a call is conditional. Guards are written as boolean expressions inside square brackets.
- An * in front of a call indicates that the call can be performed multiple times.
  - This is often followed by a guard representign the loop condition.

. . . .

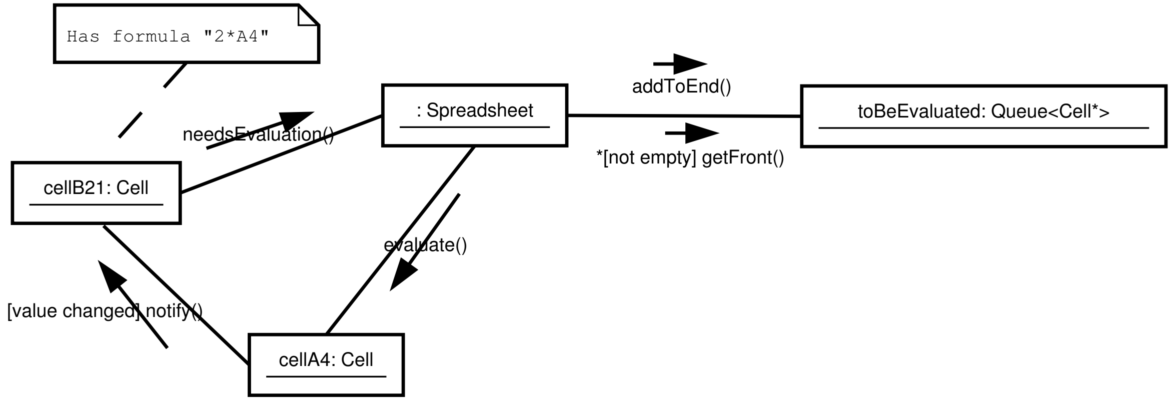
19.1.2 Sequencing

Sequencing

Collaboration diagrams work well in situations like this where the focus is on *what* the relevant messages are.



- But if the order of the messages is important to understanding the collaboration, then something like this



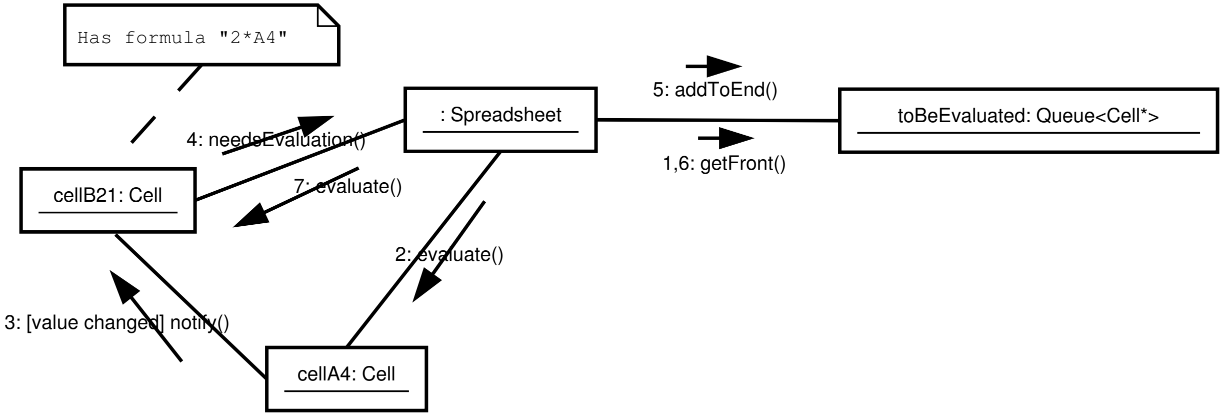
might not suffice.

. . . .

Simple Sequencing

We can try to make things clearer by adding *sequence numbers*.

- The easiest way to do this is to simply order the calls in time.



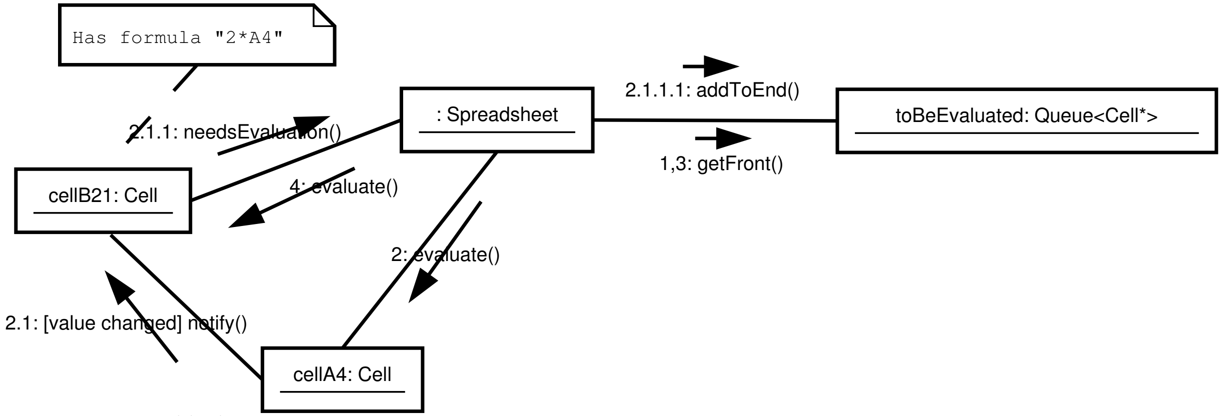
This can help a lot.

- Note, however, that it's far from clear just which of these calls come from which other calls.

. . . .

Nested Sequencing

Another possibility is to use nested sequencing calls, in which "x.y" indicates the yth step from within call# x.



This gets messy very quickly, however.

. . . .

Collaboration Diagrams do not Scale Up

If we really want to get a handle on both the order and the cause of messages, we are better off using sequence diagrams.

. . . .

19.2 Sequence Diagrams

Sequence Diagrams

These diagrams attempt to show, for some specific use-case or some common interaction:

- what objects are involved
- what operations are involved, and on which objects
- what the sequence of operations is

. . . .

19.2.1 Sequence Diagram Components

Sequence Diagram Components

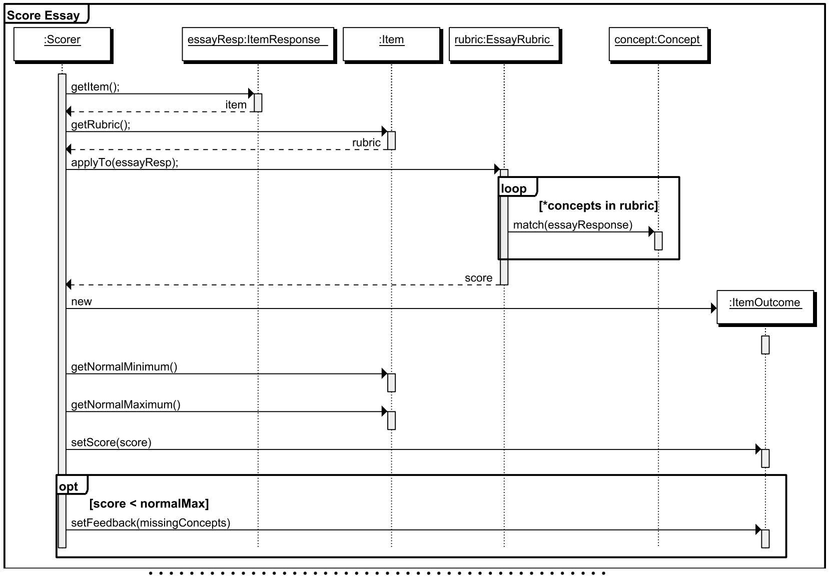
Our domain or analysis models view the world as objects that interact by exchanging messages.

- *Sequence diagrams* allow us to demonstrate that our model suffices to represent a use case by mapping the steps of the use-case into specific messages (function calls) from one object to another.
  - Although sequence diagrams can be used to illustrate any interesting *collaborations* (sequences of related messages) among our objects, we most often draw one sequence diagram for each use case.

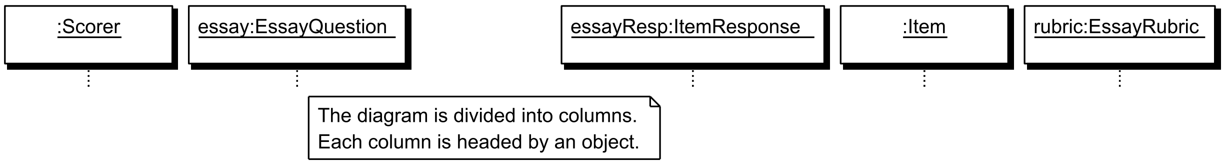
. . . .

The Score Essay Use Case

Here is a sequence diagram for our "Score Essay" use-case. To help understand this, we'll take this apart, one element at a time, looking at what each one shows us.



Seq Diagrams: Objects


:Scorer
essay:EssayQuestion
essayResp:ItemResponse
:Item
rubric:EssayRubric
The diagram is divided into columns.
Each column is headed by an object.

- A sequence diagram is composed of a number of columns, each headed by an object.
    - Human objects are often indicated by a stick figure. In general, objects are shown as rectangles.
- Note that, unlike in the class relationship diagrams, the rectangles here denote individual *objects*, not classes.

If it seems confusing to use rectangles for both, there is a consistent way to tell them apart. In UML, the general form for an object declaration is

$$objectName : className$$

For example, `chkTrans:Transaction` indicates that we have an object named “chkTrans” of type “Transaction”.

Quite often, we know we want to discuss an object, but don’t really care to give it a name. For example, in all of our use-cases so far, there has been only a single checkbook. We know that it will be a member of the `Checkbook` class, but if we actually give it a name, the reader will think that’s something they are supposed to remember. Then, if we never actually use that name, we’ve distracted the reader for no good reason.

So UML allows to drop the object name:

$$objectName : className$$

as we have done for several of the objects in the diagram above, including the checkbook.

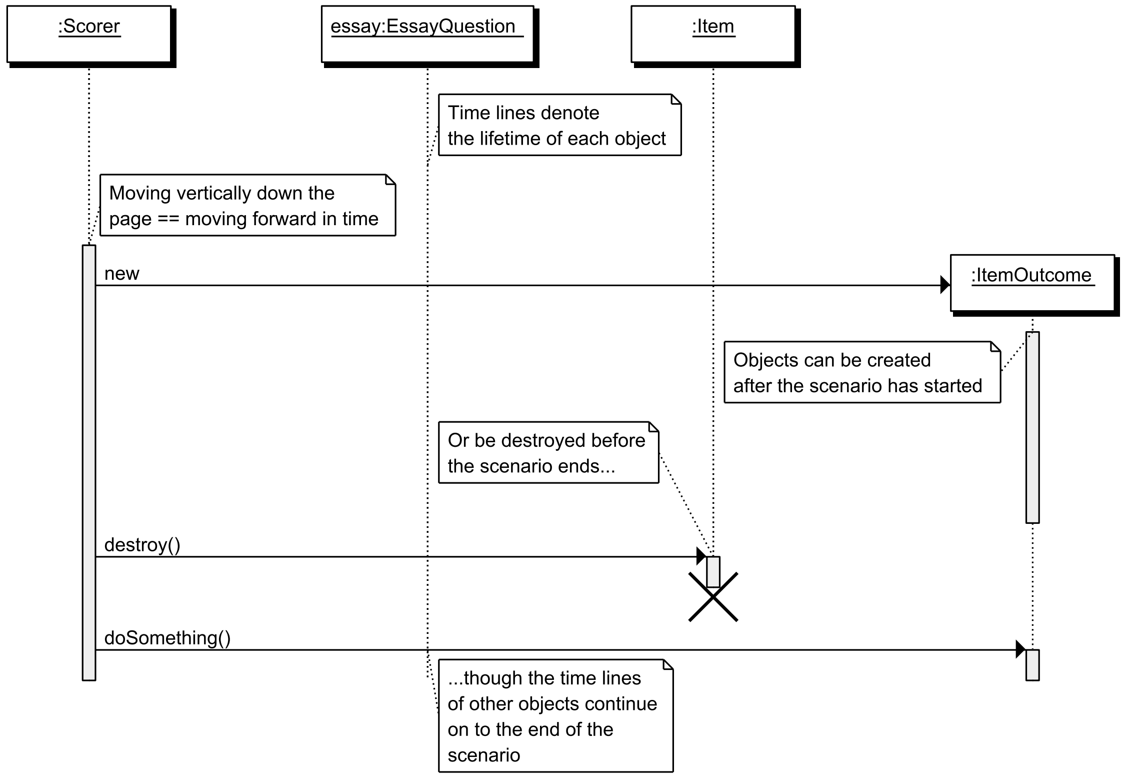
The colon (:) in front to the remaining name is our cue that

1. This is still a representation of an object, not of a class, and
2. The name that follows is a class name.

In essence, we have an anonymous object of a known class.

. . . .

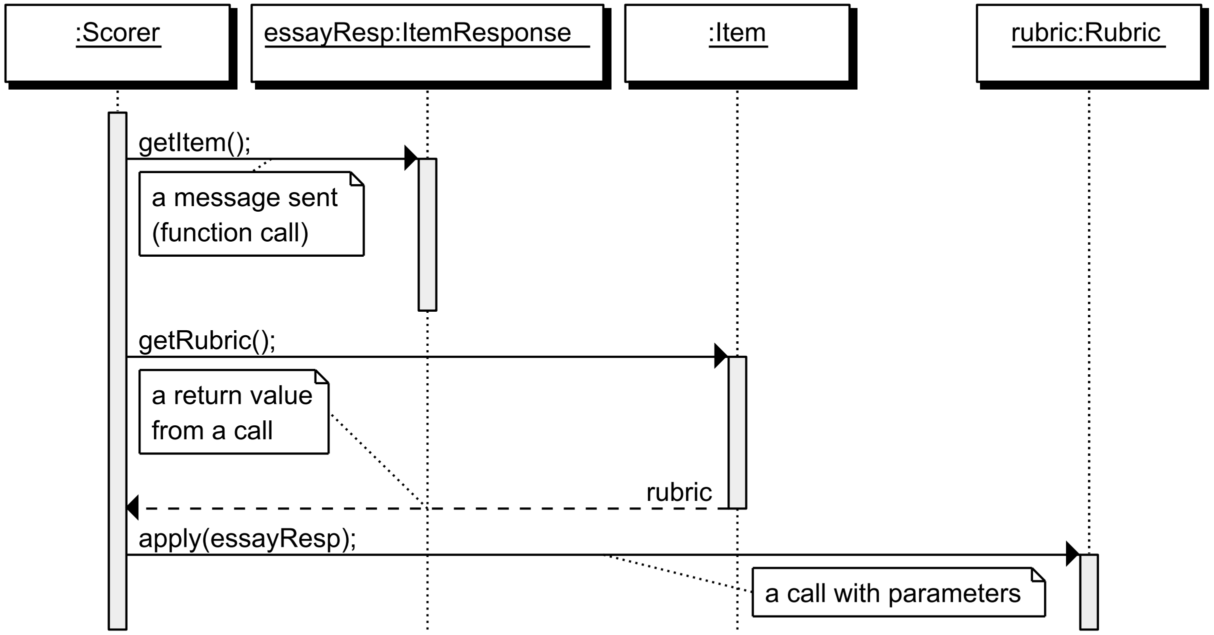
Time Lines


:Scorer
essay:EssayQuestion
:Item
Time lines denote the lifetime of each object
Moving vertically down the page == moving forward in time
new
:ItemOutcome
Objects can be created after the scenario has started
Or be destroyed before the scenario ends...
destroy()
doSomething()
...though the time lines of other objects continue on to the end of the scenario

The dashed lines are called ``time lines”.

- As we move from the top of the diagram towards the bottom, we are moving forward in time.
- Something that is drawn below a particular event actually occurs after it in our scenario.
- If an object is created/destroyed as part of a scenario, it will be shown as a shorter time line. We’ll see examples of this later.

Messages



- Each solid arrow denotes the passing of a message from one object to another.

  Or alternatively, a function call made by some code associated with the first object, calling a member function of the second object.

Parameters and Return Values

As with any function calls, these can include parameters.

- Functions can also have return values, shown as an arrow with a dashed line.
  - Normally, show these only when they are important to the understanding of a diagram (e.g., they match one of the named objects)

. . . .

Sanity Check 1

Important Sanity Check: If you draw an arrow from one object of type `C` to an object of type `T`, and you label that arrow `foo`, then the class `T` *must* have a function named `foo` listed as one of its member functions.

- If your CRC cards are still valid (eventually we abandon them as we move along in the analysis process) then `foo` should be a responsibility of `T` and `T` should be a collaborator of `C`.

. . . .

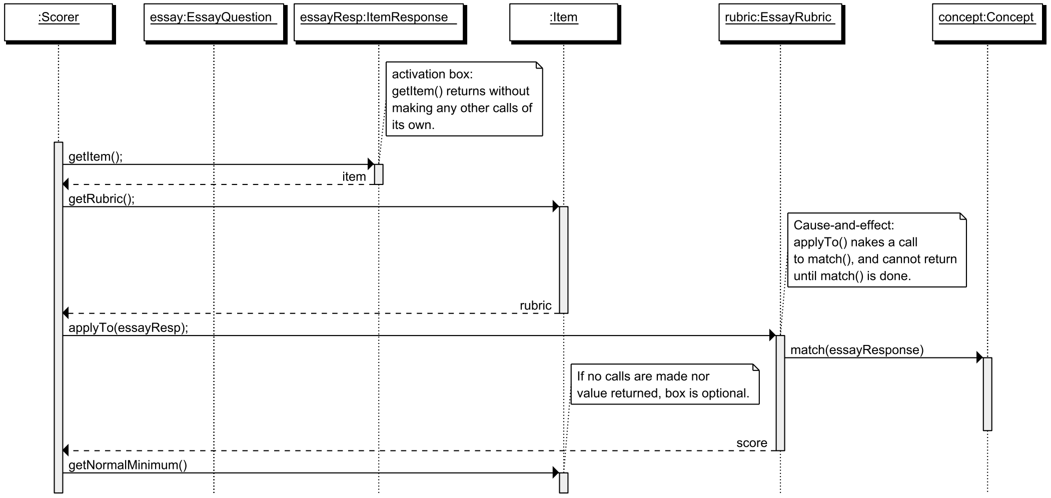
Activations

An *activation* of a function is the information associated with a particular call to that function, including all parameters, local variables, etc.

- If a function is recursive (calls itself or calls other functions that eventually call it), it can have multiple activations in memory at any given time.

. . . .

Activation Boxes


:Scorer
essay:EssayQuestion
essayResp:ItemResponse
:Item
rubric:EssayRubric
concept:Concept
activation box: getItem() returns without making any other calls of its own.
getItem();
item
getRubric();
Cause-and-effect: applyTo() nakes a call to match(), and cannot return until match() is done.
rubric
applyTo(essayResp);
match(essayResponse)
If no calls are made nor value returned, box is optional.
score
getNormalMinimum()

Sequence diagrams are all about time, so we sometimes need to indicate just how long a function call is active (i.e., how long we are executing its body or executing other function bodies called from it).

- An activation box marks off the time from the start of execution of a function body to when we return from that body to its caller.

. . . .

When to Show Activation Boxes

- You *must* show activation boxes if a the body of a called function makes other calls of its own, or if a function has a return arrow.
  - Activation boxes are essential to demonstrating cause-and-effect within the model

. . . .

Now, a sequence diagram indicates the flow of time, but it's not a strict graph. We can't say that one inch of vertical space equals some number of seconds. All that we can say is that if A is drawn below B, than A occurs after B.

Look, then at the `applyTo` call from the scorer to the rubric. You can see an activation box starting there, indicating the start of the function body of `EssayRubric::applyTo`. A little time after that, you see an arrow emerging from that box. That arrow indicates that this function body of `applyTo` eventually calls `comcept.match(essayresponse)`. There you see another function body (of `match`) start to execute.

The particular drawing tool that I use insists on putting short activation boxes at the end of every call. If there are no further outgoing calls (e.g., `getNormalMinimum()`), these boxes are optional. A message arrow with no activation box on the end simply indicates that the function call issues no interesting messages of its own.

A function body that *does* issue a message *must* show an activation box so that we can see whence that message comes.

More Sanity Checks

Sanity check 2: An incoming arrow to an activation box must connect to the very top of that box.

- We do not have psychic functions that suddenly start executing themselves in anticipation of being called at some time in the future.

Sanity check 3: Every activation box must either

- have exactly one incoming arrow, or
- have no incoming arrows but belong to an autonomous object (e.g., a human) that initiates a task spontaneously (e.g., the scorer in this scenario).

. . . .

More Sanity Checks

Sanity check 4: If a function body of `foo` calls a function `bar`, the activation box of `bar` must end *above* the end of the activation box of `foo`.

That's just the way functions work.

- If `foo` calls `bar`, then `foo` can't actually do *anything* (including returning to its own caller) until *after* `bar` returns.

Sanity check 5: If an activation box has an outgoing "return value" arrow, that arrow must emerge from the very bottom of the box.

- Function bodies do not hang around chatting after they return control to their caller.

. . . .

More Sanity Checks

Sanity check 6: If an activation box has an outgoing "return value" arrow, that arrow must point back to the caller of that function activation.

- That's just the way functions work!

. . . .

Activations Overlap (nest) in Time

The call/activation box symbols can easily accommodate simultaneous activations of different functions on the same object.

Example

Suppose we have a spreadsheet in which cell B3 contains the fomula "A1+1" and that a new formula has recently been placed in A1. A spreadsheet is in the midst of a call to

```
// Re-evaluate all cells in the spreadsheet
int evaluateAll()    {
  while(moreToEvaluate())
     partialEvaluate();
  return evaluationCounter;
}
```

. . . .

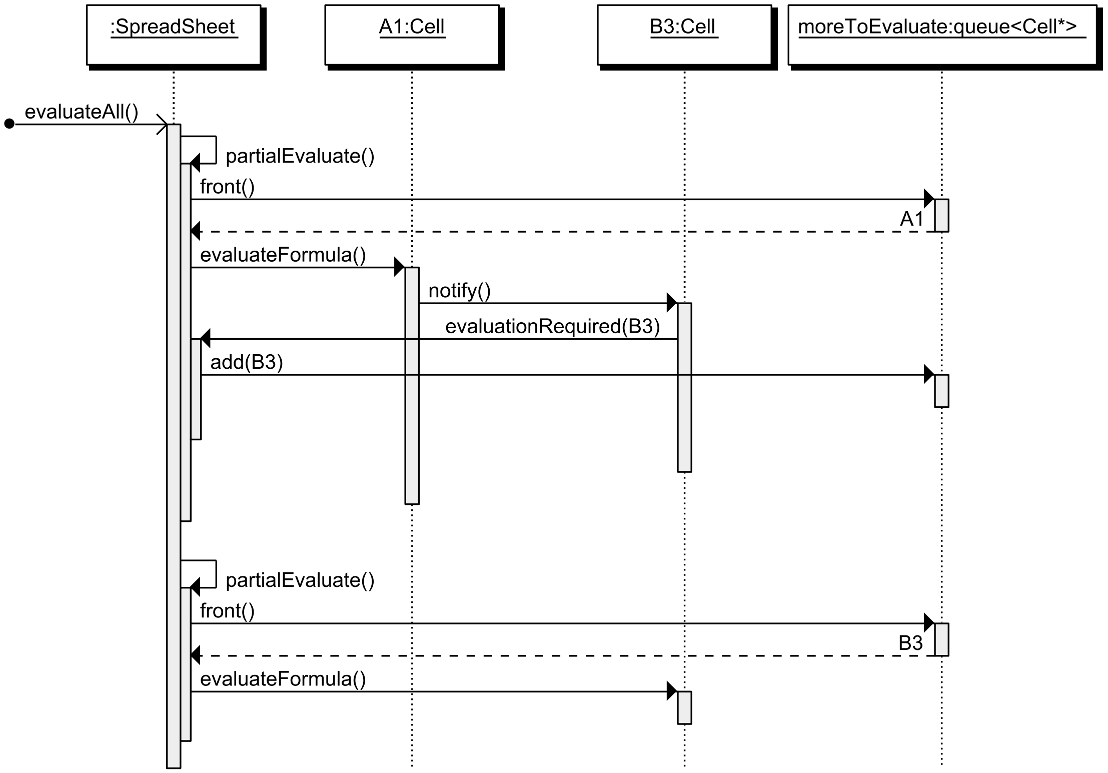
Example

Example (cont.)

- Each call to `partialEvaluate` causes one cell (e.g., A1) to be removed from a queue *moreToEvaluate* and then told to evaluate its formula.
    - If that evaluation changes the value in A1, it notifies any cells that are observing it (e.g., B3).
    - Those cells tell the spreadsheet that they require evaluation, and the spreadsheet adds them to the queue.
- On a subsequent pass around the loop, the spreadsheet makes another call to partialEvaluate that results in B3 being pulled from the queue and evaluated.

. . . .

Example: Seq Diagram

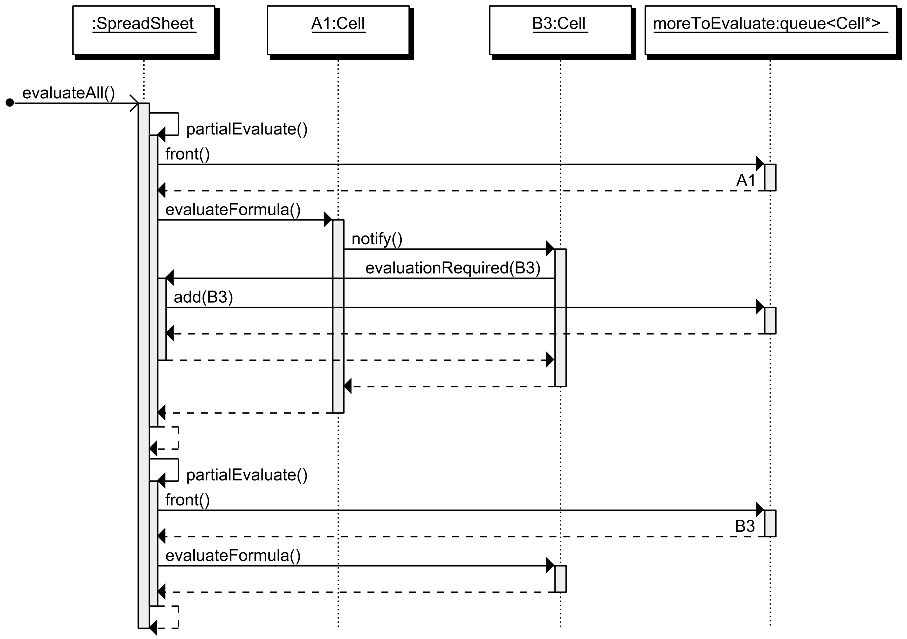


Observations:

- The call to partialEvaluate() is a call from one member function of an object to another member function of the same object.
  - We often choose not to show these because they really have little to do with the idea of whther the public interface fo a class is OK.

  - Instead, usually concentrate on inter-object messages.

- We have "unrolled" the loop in this diagram because I wanted to show the specific cells being manipulated each time around the loop.
  - We'll see other approaches to handling loops shortly.
- Note the stacking of the activation boxes on the SpreadSheet timeline to illustrate nested periods of time.
- If you have trouble following the sequence of calls and returns, you can add in the return arrows from the bottom of each activation box:



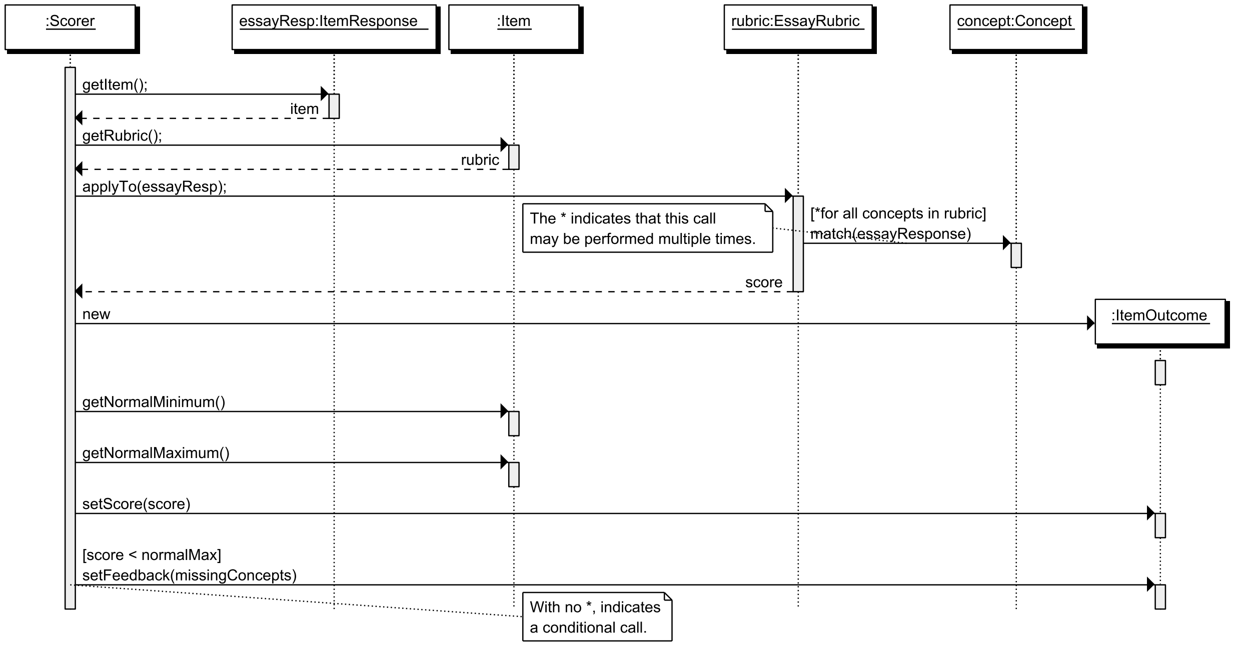
Personally, I find the extra arrows distracting and don't recommend them for general use.

19.2.2 Control Flow

Guards

Guards are conditions attached to messages to indicate conditional execution or loops.

- Guards are OK if only one call is repeated/conditional
- They become confusing when a sequence of messages are affected.



Frames

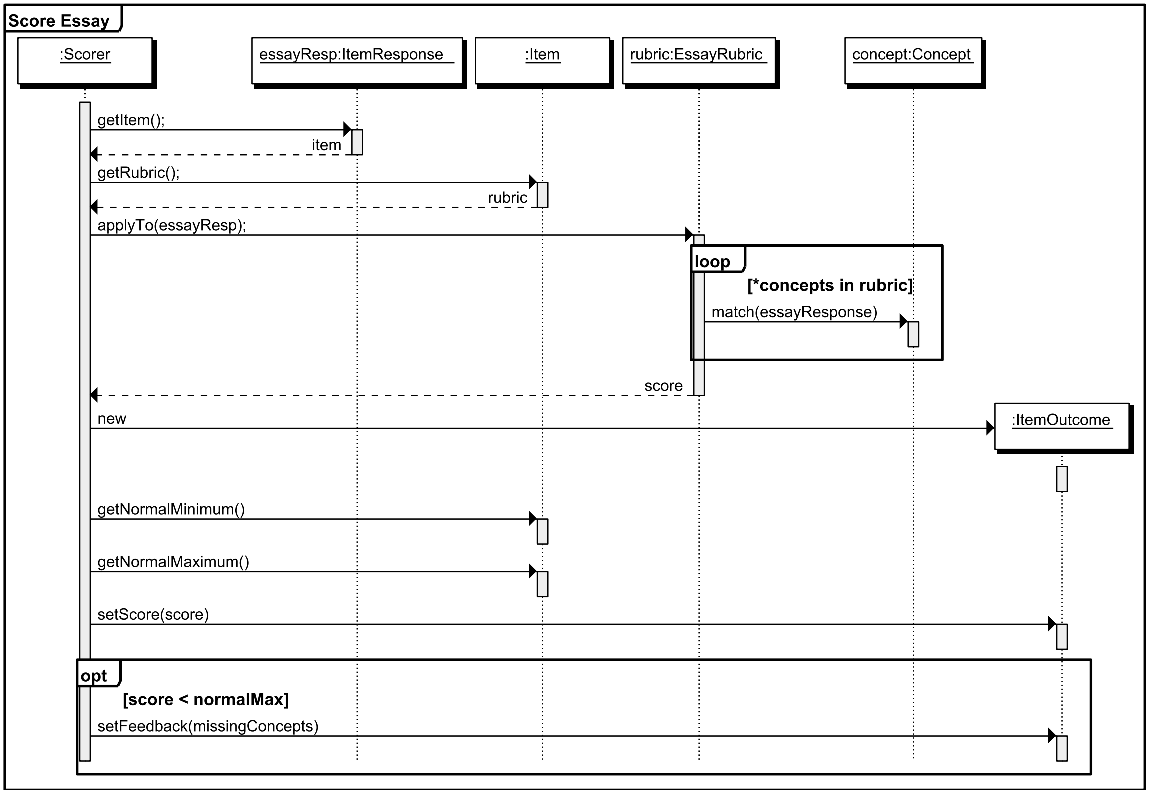
UML 2.0 introduced *frames* to group messages

Most common varieties (identified by label):

- diagram label
- loop: repeated messages
- opt: conditional messages
- alt: if-then-else like construct
- ref: reference to another diagram, becomes a "black box" in this diagram

. . . .

Options and Loops

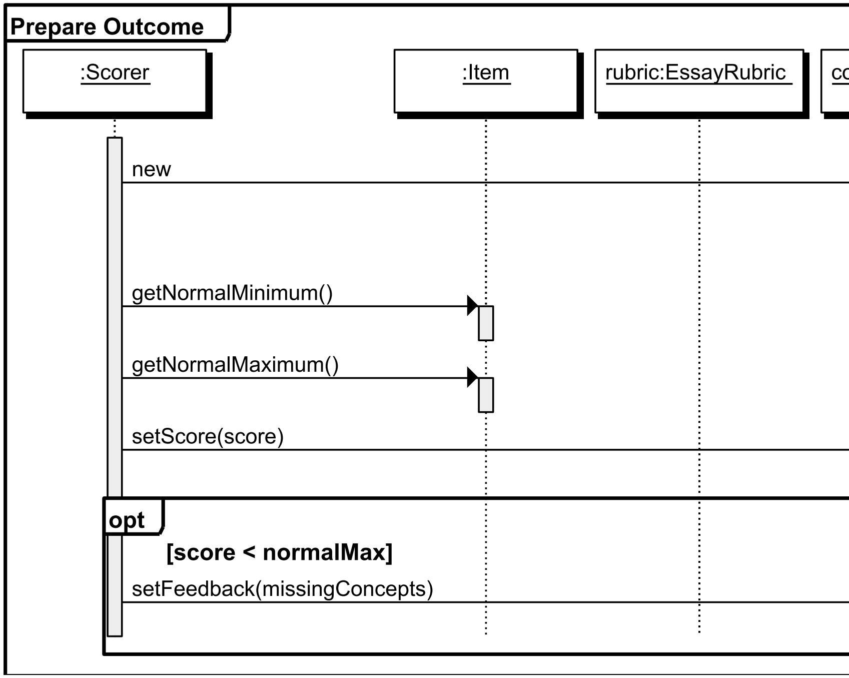


- Grouping is indicated very naturally by the area of the enclosing rectangle. We can nest frames if we need to.
- We use guard expressions (in the [ ]) to indicate loop and option conditions.

. . . .

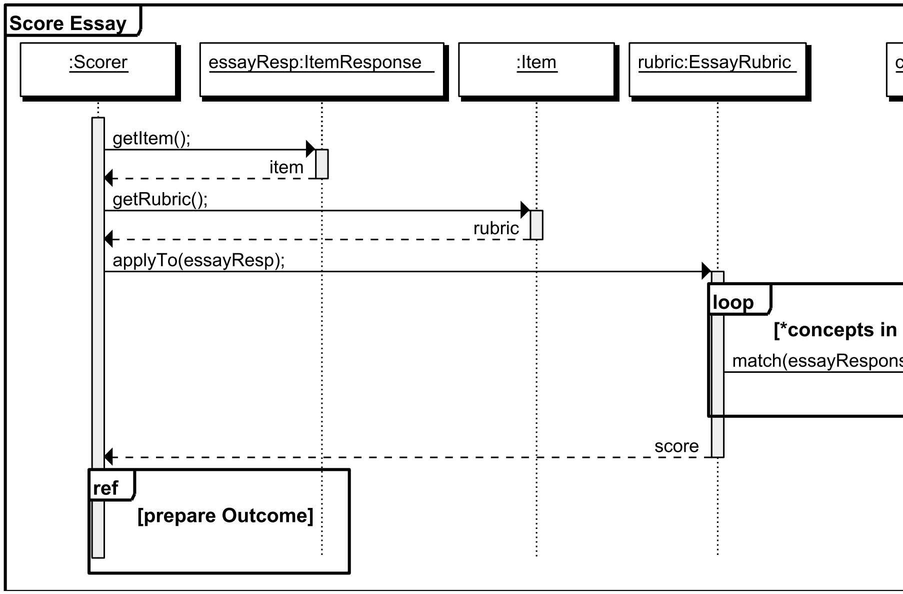
Reference Frames

We use an outer frame to name our diagram



Referring to Other Frames

And then can reference it in other diagrams



- These allow you to break up sequence diagrams into pieces and have some diagrams "include" or refer to other diagrams.

. . . .

Part VI

Java

Chapter 20

First Impressions for a C++ Programmer

Moving from C++ to Java

- First, the good news…
  - You already know most of the language
    - Syntax is largely the same as C++
    - Semantics are similar
- Then, the bad news…
  - The "standard libraries" of Java and C++ are very, very different

. . . .

I'm not going to lecture on the basics

- You already know most of them

- Readings from the course Outline page take you to CS382 material
  - do the readings
  - and the labs
- In this lesson, I'll highlight the differences between Java and C++ that you will need to get started
- Next lesson, I'll begin focusing on OO issues in Java

. . . .

20.1 Program Structure

Hello Java

The traditional starting point:

```
public class HelloWorld {

  public static void main (String[] argv)
  {
    System.out.println ("Hello, World!");
  }
}
```

- Why is `main()` inside a class?
  - Because Java has no standalone functions.
  - *All* functions must be inside a class.
- Any class with a "public static void" main function taking an array of *String*s cane be executed.

. . . .

Class Syntax

C++

```
class MailingList {
private:
   struct Node {
      Contact data;
      Node* next;
   };
   Node* first;
public:
   MailingList()
   {first = 0;}
     ⋮
};
```

Java

```
public class MailingList {
   private class Node {
      Contact data;
      Node next;
   }
   private Node first;
public MailingList()
  {first = null;}
     ⋮
}
```

- `public` and `private` are labels for each declaration, not names of "regions"
- No trailing semi-colon

. . . .

Packages

A Java *package* is like a C++ *namespace*:

- A container of classes and other, smaller packages/namespaces

C++

```
namespace myUtilities {
  class Randomizer {
     ⋮
  };
};
```

Java

```
package myUtilities;
class Randomizer {
   ⋮
}
```

- Becomes part of the *fully qualified name* of a class

C++	Java
`myUtilities::Randomizer r;`	`myUtilities.Randomizer r;`

. . . .

We Can Have Short-Cuts

C++

```
using myUtilities::Randomizer
   ⋮
Randomizer r;
```

Java

```
import myUtilities.Randomizer;
   ⋮
Randomizer r;
```

. . . .

We Can Have Shorter Short-Cuts

C++

```
using namespace myUtilities;
   ⋮
Randomizer r;
```

Java

```
import myUtilities.*;
   ⋮
Randomizer r;
```

. . . .

Cultural Difference

- C++ programmers rarely invent their own namespaces

  - And use `using` to circumvent `std::`

- Java programmers frequently invent packages
  - Often multiple packages in a single project
  - Leaving things in the untitled default package is considered the sign of a beginner

....

20.2 Program Structure == File Structure

Class == File

- In C++, I can put a class *MailingList* in `student.h` and `automobile.cpp` if I want
- Not so in Java:

  A class named "Foo" must be placed in a file named `Foo.java`
  - And upper/lower case do count!

....

Classes are not Split into Two Files

- C++ distinguishes between class
  - declarations: usually placed in a header (.h file),

```
#ifndef MAILINGLIST_H
#define MAILINGLIST_H

#include <iostream>
#include <string>
```

```
#include "contact.h"

/**
   A collection of names and addresses
*/

class MailingList
{
public:
  MailingList();
  MailingList(const MailingList&);
  ~MailingList();

  const MailingList& operator= (const MailingList&);

  // Add a new contact to the list
  void addContact (const Contact& contact);

  // Remove one matching contact
  void removeContact (const Contact&);
  void removeContact (const Name&);

  // Find and retrieve contacts
  bool contains (const Name& name) const;
  Contact getContact (const Name& name) const;


  // combine two mailing lists
  void merge (const MailingList& otherList);
  // How many contacts in list?
  int size() const;


  bool operator== (const MailingList& right) const;
  bool operator< (const MailingList& right) const;

private:

  struct ML_Node {
    Contact contact;
    ML_Node* next;

    ML_Node (const Contact& c, ML_Node* nxt)
      : contact(c), next(nxt)
    {}
  };

  int theSize;
  ML_Node* first;
  ML_Node* last;

  // helper functions
  void clear();
  void remove (ML_Node* previous, ML_Node* current);

  friend std::ostream& operator<< (std::ostream& out, const MailingList& addr);
};

// print list, sorted by Contact
std::ostream& operator<< (std::ostream& out, const MailingList& list);
```

```


#endif
```

and

– definitions: usually placed in a compilation unit (.cpp file)

```
#include <cassert>
#include <iostream>
#include <string>
#include <utility>

#include "mailinglist.h"



using namespace std;
using namespace rel_ops;




MailingList::MailingList()
  : first(NULL), last(NULL), theSize(0)
{}

MailingList::MailingList(const MailingList& ml)
  : first(NULL), last(NULL), theSize(0)
{
  for (ML_Node* current = ml.first; current != NULL;
       current = current->next)
    addContact(current->contact);
}

```

```
MailingList::~MailingList()
{
  clear();
}

const MailingList& MailingList::operator= (const MailingList& ml)
{
  if (this != &ml)
    {
      clear();
      for (ML_Node* current = ml.first; current != NULL;
       current = current->next)
    addContact(current->contact);
    }
  return *this;
}


// Add a new contact to the list
void MailingList::addContact (const Contact& contact)
{
  if (first == NULL)
    { // add to empty list
      first = last = new ML_Node(contact, NULL);
      theSize = 1;
    }
  else if (contact > last->contact)
    { // add to end of non-empty list
      last->next = new ML_Node(contact, NULL);
      last = last->next;
      ++theSize;
    }
  else if (contact < first->contact)
    { // add to front of non-empty list
      first = new ML_Node(contact, first);
      ++theSize;
    }
  else
    { // search for place to insert
      ML_Node* previous = first;
      ML_Node* current = first->next;
      assert (current != NULL);
      while (contact < current->contact)
    {
      previous = current;
      current = current->next;
      assert (current != NULL);
    }
      previous->next = new ML_Node(contact, current);
      ++theSize;
    }
}


// Remove one matching contact
void MailingList::removeContact (const Contact& contact)
{
  ML_Node* previous = NULL;
  ML_Node* current = first;
  while (current != NULL && contact > current->contact)
    {
      previous = current;
      current = current->next;
    }
  if (current != NULL && contact == current->contact)
    remove (previous, current);
}


void MailingList::removeContact (const Name& name)
{
  ML_Node* previous = NULL;
  ML_Node* current = first;
  while (current != NULL
     && name > current->contact.getName())
    {
      previous = current;
      current = current->next;
    }
  if (current != NULL
     && name == current->contact.getName())
    remove (previous, current);
}



// Find and retrieve contacts
bool MailingList::contains (const Name& name) const
{
  ML_Node* current = first;
  while (current != NULL
     && name > current->contact.getName())
    {
      previous = current;
      current = current->next;
    }
  return (current != NULL
      && name == current->contact.getName());
}


Contact MailingList::getContact (const Name& name) const
{
  ML_Node* current = first;
  while (current != NULL
     && name > current->contact.getName())
    {
      previous = current;
      current = current->next;
    }
  if (current != NULL
      && name == current->contact.getName())
    return current->contact;
  else
    return Contact();
}




// combine two mailing lists
void MailingList::merge (const MailingList& anotherList)
```

```
{
  // For a quick merge, we will loop around, checking the
  // first item in each list, and always copying the smaller
  // of the two items into result
  MailingList result;
  ML_Node* thisList = first;
  const ML_Node* otherList = anotherList.first;
  while (thisList != NULL and otherList != NULL)
    {
      if (thisList->contact < otherList->contact)
    {
      result.addContact(thisList->contact);
      thisList = thisList->next;
    }
      else
    {
      result.addContact(otherList->contact);
      otherList = otherList->next;
    }
    }
  // Now, one of the two lists has been entirely copied.
  // The other might still have stuff to copy. So we just copy
  // any remaining items from the two lists. Note that one of these
  // two loops will execute zero times.
  while (thisList != NULL)
    {
      result.addContact(thisList->contact);
      thisList = thisList->next;
    }
  while (otherList != NULL)
    {
      result.addContact(otherList->contact);
      otherList = otherList->next;
    }
  // Now result contains the merged list. Transfer that into this list.
  clear();
  first = result.first;
  last = result.last;
  theSize = result.theSize;
  result.first = result.last = NULL;
  result.theSize = 0;
}

// How many contacts in list?
int MailingList::size() const
{
  return theSize;
}


bool MailingList::operator== (const MailingList& right) const
{
  if (theSize != right.theSize) // (easy test first!)
    return false;
  else
    {
      const ML_Node* thisList = first;
      const ML_Node* otherList = right.first;
      while (thisList != NULL)
    {
      if (thisList->contact != otherList->contact)
        return false;
      thisList = thisList->next;
      otherList = otherList->next;
    }
      return true;
    }
}


bool MailingList::operator< (const MailingList& right) const
{
  if (theSize < right.theSize)
    return true;
  else
    {
      const ML_Node* thisList = first;
      const ML_Node* otherList = right.first;
      while (thisList != NULL)
    {
      if (thisList->contact < otherList->contact)
        return true;
      else if (thisList->contact > otherList->contact)
        return false;
      thisList = thisList->next;
      otherList = otherList->next;
    }
      return false;
    }
}


// helper functions
```

```
void MailingList::clear()
{
  ML_Node* current = first;
  while (current != NULL)
    {
      ML_Node* next = current->next;
      delete current;
      current = next;
    }
  first = last = NULL;
  theSize = 0;
}


void MailingList::remove (MailingList::ML_Node* previous,
       MailingList::ML_Node* current)
{
  if (previous == NULL)
    { // remove front of list
      first = current->next;
      if (last == current)
    last = NULL;
      delete current;
    }
  else if (current == last)
    { // remove end of list
      last = previous;
      last->next = NULL;
      delete current;
    }
  else
```

```
    { // remove interior node
      previous->next = current->next;
      delete current;
    }
  --theSize;
}


// print list, sorted by Contact
std::ostream& operator<< (std::ostream& out, const MailingList& list)
{
  MailingList::ML_Node* current = list.first;
  while (current != NULL)
    {
      out << current->contact << "n";
      current = current->next;
    }
  out << flush;
  return out;
}


book1.setTitle(''bogus title'');
assert (book1.getTitle() == ''bogus title'');

book2 = book1;
assert (book1 == book2);
book1.setTitle(''bogus title 2'');
assert (! (book1  == book2));
```

```
catalog.add(book1);
assert (catalog.firstBook() == book1);>




string s1, s2;
cin >> s1 >> s2;
if (s1 < s2)       ''abc'' < ''def''
                       ''abc'' < ''abcd''


    x y

Exactly one of the following is true for any x and y
    x == y
    x < y
    y < x

 namespace std{

   namespace relops {
template <class T>
bool operator!= (T left, T right)
{
  return !(left == right);
}



template <class T>
bool operator> (T left, T right)
{
  return (right < left);
}




     using namespace std::relops;
```

- Java puts the entire class into one file

```
package mailinglist;

/**
 * A collection of names and addresses
 */
public class MailingList implements Cloneable {

    /**
     * Create an empty mailing list
     *
     */
    public MailingList() {
     first = null;
     last = null;
     theSize = 0;
    }

    /**
     *  Add a new contact to the list
     *  @param contact new contact to add
     */
    public void addContact(Contact contact) {
     if (first == null) {
```

```
     // add to empty list
     first = last = new ML_Node(contact, null);
     theSize = 1;
     } else if (contact.compareTo(last.contact) > 0) {
     // add to end of non-empty list
     last.next = new ML_Node(contact, null);
     last = last.next;
     ++theSize;
     } else if (contact.compareTo(first.contact) < 0) {
     // add to front of non-empty list
     first = new ML_Node(contact, first);
     ++theSize;
     } else {
     // search for place to insert
     ML_Node previous = first;
     ML_Node current = first.next;
     assert (current != null);
     while (contact.compareTo(current.contact) < 0) {
     previous = current;
     current = current.next;
     assert (current != null);
     }
     previous.next = new ML_Node(contact, current);
     ++theSize;
     }
    }

    /**
     *  Remove one matching contact
     *  @param c remove a contact equal to c
     */
    public void removeContact(Contact c) {
     ML_Node previous = null;
     ML_Node current = first;
     while (current != null && c.compareTo(current.contact) > 0) {
     previous = current;
     current = current.next;
     }
     if (current != null && c.equals(current.contact))
     remove(previous, current);
    }

    /**
     * Remove a contact with the indicated name
     * @param name name of contact to remove
     */
    public void removeContact(String name) {
     ML_Node previous = null;
     ML_Node current = first;
     while (current != null && name.compareTo(current.contact.getName()) > 0) {
     previous = current;
     current = current.next;
     }
     if (current != null && name == current.contact.getName())
     remove(previous, current);

    }

    /**
     * Search for contacts
     * @param name name to search for
     * @return true if a contact with an equal name exists
     */
    public boolean contains(String name) {
     ML_Node current = first;
     while (current != null && name.compareTo(current.contact.getName()) > 0) {
     current = current.next;
     }
     return (current != null && name == current.contact.getName());
    }

    /**
     * Search for contacts
     * @param name name to search for
     * @return contact with that name if found, null if not found
     */
    public Contact getContact(String name) {
     ML_Node current = first;
     while (current != null && name.compareTo(current.contact.getName()) > 0) {
     current = current.next;
     }
     if (current != null && name == current.contact.getName())
     return current.contact;
     else
     return null;
    }

    /**
     * combine two mailing lists
     *
     */
    public void merge(MailingList anotherList) {
     // For a quick merge, we will loop around, checking the
     // first item in each list, and always copying the smaller
     // of the two items into result
     MailingList result = new MailingList();
     ML_Node thisList = first;
     ML_Node otherList = anotherList.first;
     while (thisList != null && otherList != null) {
     if (thisList.contact.compareTo(otherList.contact) < 0) {
     result.addContact(thisList.contact);
     thisList = thisList.next;
     } else {
     result.addContact(otherList.contact);
     otherList = otherList.next;
     }
     }
     // Now, one of the two lists has been entirely copied.
     // The other might still have stuff to copy. So we just copy
     // any remaining items from the two lists. Note that one of these
     // two loops will execute zero times.
     while (thisList != null) {
     result.addContact(thisList.contact);
     thisList = thisList.next;
     }
     while (otherList != null) {
     result.addContact(otherList.contact);
     otherList = otherList.next;
     }
     // Now result contains the merged list. Transfer that into this list.
     first = result.first;
     last = result.last;
     theSize = result.theSize;
    }
```

```
/**
 * How many contacts in list?
 */
public int size() {
 return theSize;
}

/**
 * Return true if mailing lists contain equal contacts
 */
public boolean equals(Object anotherList) {
 MailingList right = (MailingList) anotherList;
 if (theSize != right.theSize) // (easy test first!)
 return false;
 else {
 ML_Node thisList = first;
 ML_Node otherList = right.first;
 while (thisList != null) {
 if (!thisList.contact.equals(otherList.contact))
 return false;
 thisList = thisList.next;
 otherList = otherList.next;
 }
 return true;
 }
}

public int hashCode() {
 int hash = 0;
 ML_Node current = first;
```

```
     while (current != null) {
     hash = 3 * hash + current.hashCode();
     current = current.next;
     }
     return hash;
    }

    public String toString() {
     StringBuffer buf = new StringBuffer("{");
     ML_Node current = first;
     while (current != null) {
     buf.append(current.toString());
     current = current.next;
     if (current != null)
     buf.append("n");
     }
     buf.append("}");
     return buf.toString();
    }

    /**
     * Deep copy of contacts
     */
    public Object clone() {
     MailingList result = new MailingList();
     ML_Node current = first;
     while (current != null) {
     result.addContact((Contact) current.contact.clone());
     current = current.next;
     }
     return result;
```

```
}

private class ML_Node {
 public Contact contact;

 public ML_Node next;

 public ML_Node(Contact c, ML_Node nxt) {
 contact = c;
 next = nxt;
 }
}

private int theSize;

private ML_Node first;

private ML_Node last;

// helper functions
private void remove(ML_Node previous, ML_Node current) {
 if (previous == null) {
 // remove front of list
 first = current.next;
 if (last == current)
 last = null;
 } else if (current == last) {
 // remove end of list
 last = previous;
 last.next = null;
 } else {
     // remove interior node
     previous.next = current.next;
     }
     --theSize;
    }

}
```

- Function bodies are written immediately after their declaration
- To C++ programmers, these look like *inline* functions

. . . .

Package == Directory

- In C++, we can put our files into any directory we want, as long as we give the appropriate paths in our compilation commands
- In Java, all items that belong in package `packageName` must be stored in a directory filenamepackageName/

. . . .

Packaging and Compiling 1

Suppose we are building a Java project in `~/jproject`.
If we have a class

```
public class HelloWorld {

  public static void main (String[] argv)
  {
    System.out.println ("Hello, World!");
  }
}
```

- It would be stored in `~/jproject/HelloWorld,java`.
- The commands to compile and run this would be

```
cd ~/jproject
javac HelloWorld.java
java HelloWorld
```

. . . .

Packaging and Compiling 2

Suppose we are building a Java project in `~/jproject`.
If we have a class

```java
package Foo;

public class HelloWorld {

  public static void main (String[] argv)
  {
    System.out.println ("Hello, World!");
  }
}
```

- It would be stored in `~/jproject/Foo/HelloWorld,java`.
- The commands to compile and run this would be

```
cd ~/jproject
javac Foo/HelloWorld.java
java Foo.HelloWorld
```

. . . .

Packaging and Compiling 3

Suppose we are building a Java project in `~/jproject`.
If we have a class

```
package Foo.Bar;

public class HelloWorld {

  public static void main (String[] argv)
  {
    System.out.println ("Hello, World!");
  }
}
```

- It would be stored in `~/jproject/Foo/Bar/HelloWorld,java`.
- The commands to compile and run this would be

  ```
  cd ~/jproject
  javac Foo/Bar/HelloWorld.java
  java Foo.Bar.HelloWorld
  ```

. . . .

Is Java Just Trying to Be Annoying?

Actually, no.

- To compile a C++ program, we have to give explicit paths to *each* compilation unit in our compilation commands and we need to `#include` each header, giving the correct path to it.
- The Java compiler finds the source code of our other classes that we use by looking at
  - the fully qualified name (e.g., `MyUtilities.Randomizer`, or
  - our `import` statements

- It just follows the package/class names to find the directory and file in which the rest of our code is located.
- At run time, the loader acts similarly to find our compiled code.

. . . .

20.3 Swimming in Pointers

Primitives are Familiar

- *int, long, float, double*
  - *boolean*, not "bool"
  - *char* is 16 bits, not 8, to permit the use of Unicode
- Variables of these types behave as you would expect

  ```
  int x = 2;
  int y = x;
  ++x;
  System.out.println ("x=" + x + " y=" + y);
  ```

  prints "x=3 y=2"

. . . .

Everything Else is a Pointer

```
void foo(java.awt.Point p) {
  p.x = 1;
  java.awt.Point w = p;
  w.x = 2;
  System.out.println ("p.x=" + p.x + " w.x=" + w.x);
}
```

prints "p.x=2 w.x=2"

- Why did `p.x` change value?
  - Because *p* and *w* are references (pointers), so

    ```
    java.awt.Point w = p;
    ```

    causes *w* to point to the same value that *p* does.

. . . .

Lots of Allocation

Because all new class variables are really pointers, all new class values have to be created on the heap:

C++

```
Point p (1,2);
```

Java

```
Point p = new Point(1,2);
```

. . . .

Arrays of Pointers

C++ programmers need to be particularly careful dealing with arrays:

C++

```
int a[10];
Point* p = new Point[10];
```

Java

```
int[] b = new int[10];
Point q = new Point[10];
for (int i = 0;
      i < q.length; ++i)
   q[i] = new Point();
```

Without the loop, *q* would actually be an array of null pointers.

. . . .

Because there are so many pointers

- Sharing in Java is much more common than copying
  - If you do need a distinct copy, use the `clone()` function
    - We'll see later that this is a "standard" function on all Java objects
- It's a good thing that Java features automatic garbage collection!

. . . .

Beware of ==

- The == operator works like you would expect on primitives

```
int x = 23;
int y = 23;
if (x == y)
   System.out.println ("Of course!");
```

- But for class objects, == is comparing addresses:

```
Point p = new Point(1,2);
Point q = new Point(1,2);
if (p == q)
   System.out.println ("Not gonna happen");
else
   System.out.println ("Surprise!");
```

. . . .

equals

To compare objects to see if they have the same *contents*, use the `equals` function:

```
Point p = new Point(1,2);
Point q = new Point(1,2);
if (p.equals(q))
   System.out.println ("That's better.");
else
   System.out.println ("Pay no attention...");
```

. . . .

20.4 Exceptions

Exceptions

An *exception* is a run-time error signal.

- It may be signalled (*thrown*) by the underlying runtime system. . .
  - or by programmer-suppled code
- Programs can *catch* exceptions and *handle* them . . .
  - or let them go and allow the program to abort

. . . .

Playing Catch

Try compiling and (if successful), running each of the following:

```
import java.io.*;

/**
   Demo of a program that may throw exceptions.
   @param argv The name of a file to open for input
*/
public class OpenFile1 {

  /**
     Attempt to open a file
  */
  static void openFile (String fileName) {
    FileReader reader = new FileReader(fileName);
  }

  /**
     Attempt to open the file whose name is given in
     the first command line parmaeter
  */
  public static void main (String[] argv) {
    String fileName = argv[1];
    openFile (fileName);
  }
}
```

```
import java.io.*;

/**
   Demo of a program that may throw exceptions.
   @param argv The name of a file to open for input
*/
public class OpenFile2 {

  /**
     Attempt to open a file
  */
  static void openFile (String fileName)
    throws java.io.FileNotFoundException
  {
    FileReader reader = new FileReader(fileName);
  }

  /**
     Attempt to open the file whose name is given in
     the first command line parmaeter
  */
  public static void main (String[] argv) {
    String fileName = argv[0];
    openFile (fileName);
  }
}
```

```
import java.io.*;

/**
   Demo of a program that may throw exceptions.
   @param argv The name of a file to open for input
*/
public class OpenFile3 {

  /**
     Attempt to open a file
  */
  static void openFile (String fileName)
    throws java.io.FileNotFoundException
  {
    FileReader reader = new FileReader(fileName);
  }

  /**
     Attempt to open the file whose name is given in
     the first command line parmaeter
  */
  public static void main (String[] argv) {
      try {
      openFile (argv[0]);
      }
      catch (java.io.FileNotFoundException ex)
      {
          System.err.println ("Something is wrong with the file: " + ex);
      }
      System.out.println ("All done");
  }
}
```

using both the names of exiting and non-exiting files, or no name at all.

. . . .

Unchecked Exceptions

Exceptions come in two main kinds: checked and unchecked

- *unchecked* exceptions could arise almost anywhere
  - e.g., `NullPointerException`, `ArrayIndexOutOfBoundsException`
- functions need not declare that they might throw these

. . . .

Checked Exceptions

checked exceptions are more specialized

- include all programmer-defined exceptions
- functions *must declare* if they can throw these

. . . .

Another Cultural Difference

- C++ actually has exceptions as well.
  - Same `throw` and `try - catch` syntax
  - Can't declare what exceptions a function is known to throw
- But they are used much more widely in Java

. . . .

20.5 Corresponding Data Structures

The Java API

The Java API is huge, but well documented

- Focus initially on packages `java.lang`, `java.io`, and `java.util`
- BTW, You can use the **javadoc** to generate the same kind of documentation for your own code. Example

. . . .

Strings

C++ *std::string* : Java *java.lang.String*

- Strings in Java are *immutable*
  - You cannot change the contents of a string value
  - But you can compute a slightly different value and make a string variable point to the different value

C++

```
string s = ...;
s[s.size()/2] = 'b';
s = s + s;
```

Java

```
String s = ...;
s = s.substring(0,s.length()/2)
  + 'b'
  +  s.substring(s.length()/2+1);
s = s + s;
```

. . . .

If You Need to Change a Java String

use a *StringBuilder*

```
StringBuilder sb = new StringBuilder();
String line = input.readline();
while (line != null) {
  sb.append(line);
  sb.append("\n");
  line = input.readline();
}
String allTheText = sb.toString();
```

. . . .

vector : ArrayList

C++

```
vector<string> v;
v.push_back("foo");
cout << v[0] << endl
```

Java

```
ArrayList<String> v = new ArrayList<String>();
v.add("foo");
System.out.println (v.get(0));
```

. . . .

list : LinkedList

C++

```
list<string> L;
L.push_back("foo");
cout << L.front() << endl
```

Java

```
LinkedList<String> L = new LinkedList<String>();
L.add("foo");
System.out.println (L.getFirst());
```

. . . .

set : HashSet

C++

```
set<string> s;
s.insert("foo");
cout << s.count("foo") << endl
```

Java

```
HashSet<String> S = new HashSet<String>();
S.add("foo");
System.out.println ("" + S.contains("foo"));
```

. . . .

map : HashMap

C++

```
map<string,int> zip;
zip["ODU"] = 23529;
cout << zip["ODU"] << endl
```

Java

```
HashMap<String,Integer> zip
      = new HashMap<String,Integer>();
zip.put("ODU", 23529);
System.out.println (zip.get("ODU"));
```

. . . .

Chapter 21

Inheritance in Java

21.1 Class Inheritance

Class Inheritance

Inheritance among classes in Java works almost identically to C++. The only difference is a rather minor change in syntax. Instead of, for example,

```
class NumericValue: public Value {
```

as in C++, Java uses the reserved word `extends` to indicate the same thing:

```
class NumericValue extends Value {
```

What is different in Java is not *how* we do inheritance, but *how often*.

. . . .

In C++, if we declare a new class and do not explicitly inherit from another, our new class has no base class. But in Java, if we do not explicitly inherit from another class, then our class will implicitly inherit from a class named `Object`, more specifically, `java.lang.Object`.

This is not an empty piece of language design theory. As we will see shortly, there are very real functions declared by `Object` that we inherit and may want to use. In fact, we have seen some of them (e.g., `equals`, `clone`, and `toString`) in our earlier checklist. One reason I was able to assert that these members represent a consensus among Java programmers is simply because they are encoded into the `Object`.

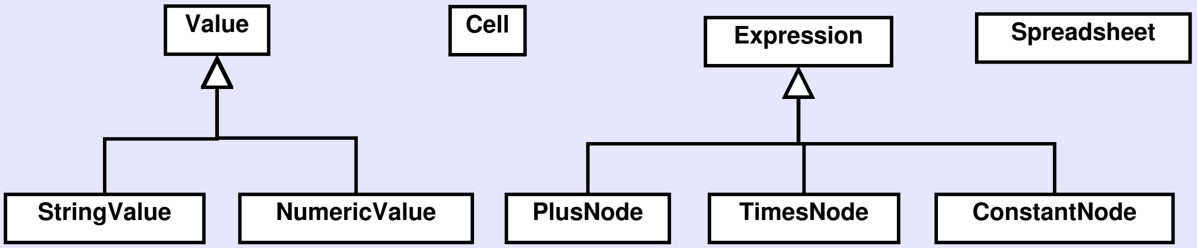
Keep in mind that, even if we do explicitly inherit from some other base class, that base class will either inherit from `Object` or from some other class that either inherits from `Object` or . . . Eventually, we *will* be inheriting from `Object`, so the members declared there will be inherited by *every* class.

21.1.1 Shrubs and Trees

Inheritance in C++

This special `Object` class contributes to a difference in style between C++ and Java. C++ programs use inheritance only rarely and when there is an obvious need for it.

Spreadsheet inheritance in C++:



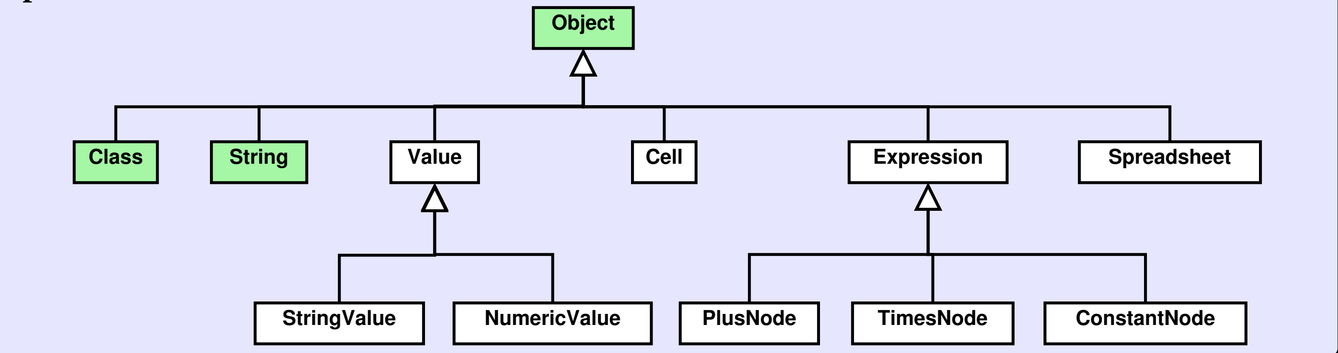
. . . .

Our spreadsheet program has inheritance, but only for a minority of closely related classes.

This style permeates C++. The std:: library has very little use of inheritance, particularly in the design of utility data structures and collections.

Inheritance in Java

Spreadsheet inheritance in Java:



In Java, our spreadsheet program has the same "topical" inheritance , but our other classes are gathered together into the common tree of Java classes.

The Java API makes a great deal more obvious use of inheritance than does C++ std.

. . . .

In fact, until the fairly recent (Java 1.5) introduction of "generics" (similar to templates in C++), it was nearly impossible to do anything useful with Java data structures without exploiting inheritance.

Comparing the two diagrams, you can see why I often say that Java programs are organized as one large inheritance tree, but C++ programs as a collection of small shrubs.

21.1.2 Inheriting from Object

Inheriting from Object

If a class declaration does not explicitly state a superclass, by default it inherits from `Object`.

What do we get from this?

Some examples if the members we inherit:

. . . .

What Can You do to Any Object?

- `protected native Object clone()`
  - Creates a new object of the same class as this object.
- `public boolean equals(Object)`
  - Compares two Objects for equality.
- `finalize()`
  - Called by the garbage collector on an object when there are no more references to the object.
- `public final Class getClass()`
  - Returns the runtime class of an object.
- `public native int hashCode()`
  - Returns a hash code value for the object.
- `public String toString()`
  - Returns a string representation of the object.
  - called implicitly as needed for conversion to string

```
System.out.println (c.getName()
                    + " has formula "
                    + c.getFormula());
```

We'll encounter other *Object* functions later.

21.2 Dynamic Binding in Java

Dynamic Binding in Java

Dynamic binding is more pervasive in Java, because

- Almost all functions in Java are, implicitly, virtual.
  - The only exceptions are functions that are explicitly marked as `static`.
- *All* non-primitive variables in Java contain references to objects on the heap.

  We don't have the distinction made in C++ between functions invoked directly on an object as opposed to functions invoked via a pointer/reference.

....

21.2.1 The Animal Example in Java

The Animal Hierarchy in Java

Same example, this time in Java. Start with the inheritance hierarchy.

```
public class Animal {
  public String eats() {return "???";}
  public name() {return "Animal";}
}

public class Herbivore extends Animal {
   public String eats() {return "plants";}
   public String name() {return "Herbivore";}
}

public class Ruminants extends Herbivore {
   public String eats() {return "grass";}
   public String name() {return "Ruminant";}
}
```

```
public class Carnivore extends Animal {
   public String eats() {return "meat";}
   public String name() {return "Carnivore";}
}
```

....

The Main Program

```
public class AnimalTest {

  private static void show (String s1, String s2) {
     System.out.println (s1 + " " + s2);
  }

  public static void main (String[]) {
     Animal a = new Animal();
     Herbivore h = new Herbivore();
     Ruminant r = new Ruminant();

     Animal paa = a;
     Animal pah = h;
     Animal par = r;

     show(a.name(), a.eats());        // AHRC ?pgm
     show(paa.name(), paa.eats()); // AHRC ?pgm
     show(h.name(), h.eats);          // AHRC ?pgm
     show(pah.name(), pah.eats()); // AHRC ?pgm
     show(par.name(), par.eats()); //AHRC ?pgm
  }
}
```

The application is structured a bit differently. Java does not allow standalone functions, so the `show` function needs to be inside a class. We assume the rest of the application code is in that same class.

Note the absence of the ∗ and & operators. Since almost everything is a pointer, special operators for dereferencing pointers and fetching addresses are not needed.

. . . .

Animal Hierarchy: a

```
public class AnimalTest {

  private static void show (String s1, String s2) {
     System.out.println (s1 + " " + s2);
  }

  public static void main (String[]) {
     Animal a = new Animal();
     Herbivore h = new Herbivore();
     Ruminant r = new Ruminant();

     Animal paa = a;
     Animal pah = h;
     Animal par = r;

     show(a.name(), a.eats());      // AHRC ?pgm
     show(paa.name(), paa.eats()); // AHRC ?pgm
     show(h.name(), h.eats);        // AHRC ?pgm
     show(pah.name(), pah.eats()); // AHRC ?pgm
     show(par.name(), par.eats()); //AHRC ?pgm
  }
}
```

. . . .

Unlike C++, Java resolves *all* function calls by dynamic binding. And, unlike C++, `a` is a pointer, whether we like it or not. So, at runtime, the program follows `a` out to the heap, discovers that it actually points to an `Animal`, and invokes the `Animal::name()` function body to print "Animal". Then it does the same for `eats()` and invokes the `Animal::eats()` function body to print "???".

Animal Hierarchy: paa

```
public class AnimalTest {

  private static void show (String s1, String s2) {
     System.out.println (s1 + " " + s2);
  }

  public static void main (String[]) {
     Animal a = new Animal();
     Herbivore h = new Herbivore();
     Ruminant r = new Ruminant();

     Animal paa = a;
     Animal pah = h;
     Animal par = r;

     show(a.name(), a.eats());      // Animal ???
     show(paa.name(), paa.eats()); // AHRC ?pgm
     show(h.name(), h.eats);        // AHRC ?pgm
     show(pah.name(), pah.eats()); // AHRC ?pgm
     show(par.name(), par.eats()); //AHRC ?pgm
  }
}
```

paa is treated just like a because, after all, they are both pointers and pointing to the same object.

. . . .

Animal Hierarchy: h

```
public class AnimalTest {

  private static void show (String s1, String s2) {
     System.out.println (s1 + " " + s2);
  }

  public static void main (String[]) {
     Animal a = new Animal();
     Herbivore h = new Herbivore();
     Ruminant r = new Ruminant();

     Animal paa = a;
     Animal pah = h;
     Animal par = r;

     show(a.name(), a.eats());      // Animal ???
     show(paa.name(), paa.eats()); // Animal ???
     show(h.name(), h.eats);        // AHRC ?pgm
     show(pah.name(), pah.eats()); // AHRC ?pgm
     show(par.name(), par.eats()); //AHRC ?pgm
  }
}
```

. . . .

At runtime, the program follows `h` out to the heap, discovers that it actually points to an `Herbivore`, and invokes the `Herbivore::name()` function body to print "Herbivore". Then it does the same for `eats()` and invokes the `Herbivore::eats()` function body to print "plants".

Animal Hierarchy: pah

```
public class AnimalTest {

  private static void show (String s1, String s2) {
     System.out.println (s1 + " " + s2);
  }

  public static void main (String[]) {
     Animal a = new Animal();
     Herbivore h = new Herbivore();
     Ruminant r = new Ruminant();

     Animal paa = a;
     Animal pah = h;
     Animal par = r;

     show(a.name(), a.eats());      // Animal ???
     show(paa.name(), paa.eats()); // Animal ???
     show(h.name(), h.eats);        // Herbivore plants
     show(pah.name(), pah.eats()); // AHRC ?pgm
     show(par.name(), par.eats()); //AHRC ?pgm
  }
}
```

....

At runtime, the program follows `pah` out to the heap, discovers that it actually points to an `Herbivore`, and invokes the `Herbivore::name()` function body to print "Herbivore". Then it does the same for `eats()` and invokes the `Herbivore::eats()` function body to print "plants".

Animal Hierarchy: par

```
public class AnimalTest {
  private static void show (String s1, String s2) {
     System.out.println (s1 + " " + s2);
  }

  public static void main (String[]) {
     Animal a = new Animal();
     Herbivore h = new Herbivore();
     Ruminant r = new Ruminant();

     Animal paa = a;
     Animal pah = h;
     Animal par = r;

     show(a.name(), a.eats());      // Animal ???
     show(paa.name(), paa.eats()); // Animal ???
     show(h.name(), h.eats);        // Herbivore plants
     show(pah.name(), pah.eats()); // Herbivore plants
     show(par.name(), par.eats()); //AHRC ?pgm
  }
}
```

. . . .

At runtime, the program follows `par` out to the heap, discovers that it actually points to a `Ruminant`, and invokes the `Ruminant::name()` function body to print "Ruminant". Then it does the same for `eats()` and invokes the `Ruminant::eats()` function body to print "grass".

Final Results

```
public class AnimalTest {
  private static void show (String s1, String s2) {
     System.out.println (s1 + " " + s2);
  }

  public static void main (String[]) {
     Animal a = new Animal();
     Herbivore h = new Herbivore();
     Ruminant r = new Ruminant();

     Animal paa = a;
     Animal pah = h;
     Animal par = r;

     show(a.name(), a.eats());      // Animal ???
     show(paa.name(), paa.eats()); // Animal ???
     show(h.name(), h.eats);        // Herbivore plants
     show(pah.name(), pah.eats()); // Herbivore plants
     show(par.name(), par.eats()); // Ruminant grass
  }
}
```

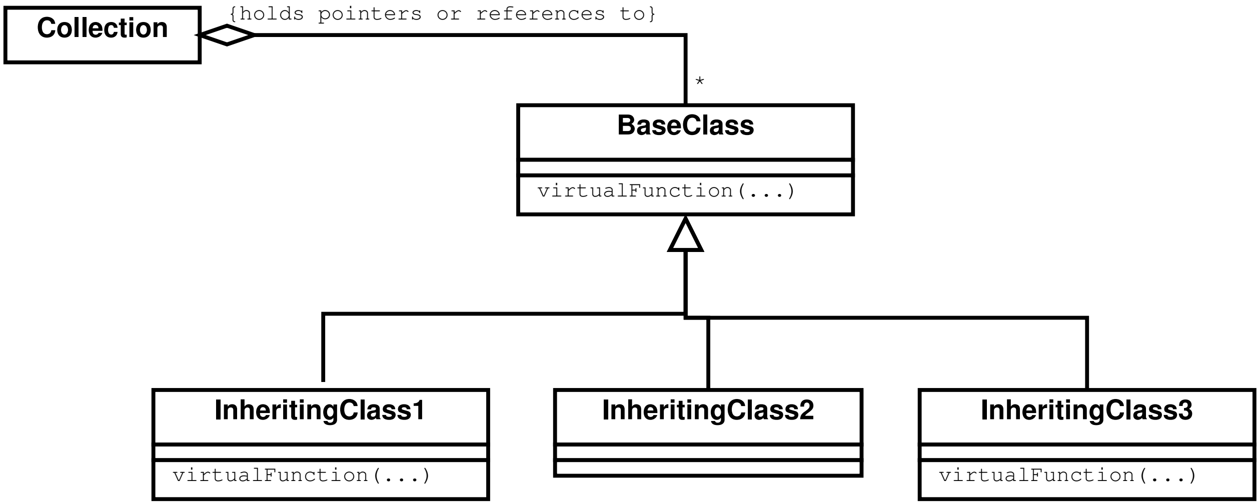
The overall output here is slightly different from that in the earlier C++ example because, in Java, the `name()` function is "virtual" and so is handled by dynamic binding.

. . . .

21.2.2 The Key Pattern of OOP in Java

The Key Pattern of OOP

Just as in C++, suppose we have an inheritance hierarchy:



and that we have a collection of (references to) the `BaseClass`

```
Collection collection;
```

Then this code:

```
BaseClass x;
for (each x in collection) {
  x.memberFunction(...);
}
```

uses dynamic binding to apply subclass-appropriate behavior to each element of a collection. Each time around the loop, we extract a reference from the collection. Thanks to subtyping, that reference could be pointing to something of type `BaseClass` or to any of its subclasses. But when we call `memberFunction` through that reference, the runtime system uses the data type of the thing pointed to determine which function body to invoke. If we have enough subclasses, we could wind up doing a different function body each time around the loop.

. . . .

Examples of the key pattern

Examples of the key pattern

There are lots of variations on this pattern. We can use almost any data structure for the collection.

. . . .

Example: arrays of Animals in Java

```
Animal[] animals = new Animal[numberOfAnimals];
   ⋮
for (int i = 0; i < numberOfAnimals; ++i)
   System.out.println (animals[i].name()
       + " " + animals[i].eats());
```

In newer versions of Java (1.5 or later), the loop above can be simplified:

```
Animal[] animals = new Animal[numberOfAnimals];
   ⋮
for (Animal a: animals)
   System.out.println (a.name() + " " + a.eats());
```

- This is an example of the “for each” style loop in Java which can be used with any array so long as you don’t actually need the index value for some reason.

. . . .

Example: Linked Lists of Animals (Java)

```
class ListNode {
   Animal data;
   ListNode next;
}
ListNode head; // start of list
   ⋮
```

```
for (ListNode current = head; current != null;
        current = current.next)
    System.out.println (current.name()
        + " " + current.eats());
```

- Notice the complete lack of * or -> operators.
  - As C++ programmers, we are used to seeing those as cues that we are dealing with pointers.

. . . .

Example: vector (ArrayList) of Animals

```
ArrayList<Animal> animals = new ArrayList<Animal>();
    ⋮
for (Animal a: animals) {
    System.out.println (a.name() + " " + a.eats());
}
```

- The `for` loop shown here is, again, an example of the Java "for each" syntax. For most Java container types, we can iterate through all elements in the container by simply declaring the loop variable (`Animal a`) and then, after a ":", naming the container (*animals*).
- Look also at the rather template-like declaration of the *animals* container. This is a fairly recent addition to Java.

. . . .

Older Style Java

Older Java code would leave out the part within the angle brackets, in which case we would have to treat animals as a container of *Object*s, not *Animal*s.

```
ArrayList animals = new ArrayList();
   ⋮
for (Object obja: animals) {
    Animal a = (Animal)obja;
   System.out.println (a.name() + " " + a.eats());
}
```

The downcast is required because the loop variable, *obja*, is of type *Object* and therefore does not support the `name()` and `eats()` functions.

....

21.2.3 Abstract Classes in Java

Abstract Classes in Java

Abstract classes work much the same in Java as in C++. The only differences are

- In Java, you must label both the class and its abstract functions. (In C++ you label only the functions.)
- Instead of putting `=0` at the end of the function, you put the keyword `abstract` in front of the function/class declaration.

....

Abstract Class Exa mple

Here, for example, is the Java version of our abstract spreadsheet value class.

```
package SpreadSheetJ.Model;



//
// Represents a value that might be obtained for some spreadsheet cell
// when its formula was evaluated.
//
// Values may come in many forms. At the very least, we can expect that
// our spreadsheet will support numeric and string values, and will
// probably need an "error" or "invalid" value type as well. Later we may
// want to add addiitonal value kinds, such as currency or dates.
//
public abstract
class Value implements Cloneable
{
    public abstract String valueKind();
    // Indicates what kind of value this is. For any two values, v1 and v2,
    // v1.valueKind() == v2.valueKind() if and only if they are of the
    // same kind (e.g., two numeric values). The actual character string
    // pointed to by valueKind() may be anything, but should be set to
    // something descriptive as an aid in identification and debugging.


    public abstract String render (int maxWidth);
    // Produce a string denoting this value such that the
    // string's length() <= maxWidth (assuming maxWidth > 0)
    // If maxWidth==0, then the output string may be arbitrarily long.
    // This function is intended to supply the text for display in the
    // cells of a spreadsheet.

    public String toString()
    {
    return render(0);
    }

    public boolean equals (Object value)
    {
    Value v = (Value)value;
    return (valueKind() == v.valueKind()) && isEqual(v);
    }


    abstract boolean isEqual (Value v);
    //pre: valueKind() == v.valueKind()
    //  Returns true iff this value is equal to v, using a comparison
    //  appropriate to the kind of value.

}
```

. . . .

21.3 Interfaces In Java

Interfaces In Java

Java offers an alternate, closely related, mechanism for relating classes to one another, the *interface*.

- An *interface* declares a related set of
  - member function declarations
  - constant values
- Classes may be declared to *implement* an interface independently of where they are in the inheritance hierarchy.

. . . .

Example: AudioClip

```
public interface AudioClip {
    /**
     * Starts playing this audio clip. Each time this method is called,
      * the clip is restarted from the beginning.
      */
    void play();

    /**
     * Starts playing this audio clip in a loop.
     */
    void loop();

    /**
     * Stops playing this audio clip.
     */
    void stop();
}
```

. . . .

Example: Cloneable

Signals that a class has a working `clone()` function.

- Otherwise, `Object.clone()` will throw an exception.

```
package java.lang;
public interface Cloneable {
   public Object clone();
}
```

. . . .

21.3.1 Interface Implementation is Not Inheritance

Interface Implementation is Not Inheritance

- You cannot inherit variables from an interface.
- You cannot inherit method implementations (function bodies) from an interface.
- The interface hierarchy is independent of a the class hierarchy.
- A Java class may implement many different interfaces, but can only inherit from one superclass.

. . . .

So What is it, Then?

A class that implements an interface can be used wherever that interface is "expected". That's pretty much our definition of *subtyping*.

. . . .

21.3.2 Example: Sorting in Java

Example: Sorting in Java

Suppose we wanted to provide a class that collected a number of useful sorting algorithms.

- All sorting algorithms require the ability to compare objects.
- But class `Object` has no comparison function except `equals`.

How do we tell potential users of our sorting routines how to provide comparison functions that we can use?

. . . .

A C++ style Solution - Inheritance

```
abstract class Comparable {
  public abstract boolean comesBefore (Object o);
}
```

One solution is to define the "comparable" protocol as a class.

Then we can write our sorting functions to work on objects of this `Comparable` class (which we really never expect to ever see) or of any subtype of `Comparable`:

. . . .

Calling on a Comparable Class

```
class Sorting {

  public static void
    insertionSort (Comparable[] array)
  {
     for (int i = 1; i < array.length; ++i) {
       Comparable temp = array[i];
       int p = i;
       while ((i > 0)
         && temp.comesBefore(array[p-1])) {
          array[p] = array[p-1];
          p--;
       }
       array[p] = temp;
    }
  }
    ⋮
}
```

. . . .

Extending Comparable

Here's an example of how we might declare a class that overrides `comesBefore` to provide a "sensible" implementation we can use for sorting.

```
class Student extends Comparable
{
  String name;
  String id;
  double gpa;
  String school;

  boolean comesBefore(Object o)
  {
    return gpa > ((Student)o).gpa;
  }

}
```

In this case, we can sort students by grade point average.

. . . .

A Closer Look

The downcast in our application

```
class Student extends Comparable
{
  String name;
  String id;
  double gpa;
  String school;

  boolean comesBefore(Object o)
  {
    return gpa > ((Student)o).gpa;
  }

}
```

is ugly (but required) to match:

```
abstract class Comparable {
  public abstract boolean comesBefore (Object o);
}
```

. . . .

Making Comparable Generic

We can get rid of it by making *Comparable* *generic*, so that it knows what kind of objects it is actually comparing:

```
abstract class Comparable<T extends Object> {
  public abstract boolean comesBefore (T t);
}
```

and then letting the *Student* class pass that info along:

```
class Student extends Comparable<Student>
{
   String name;
   String id;
   double gpa;
   String school;

   boolean comesBefore(Student s)
   {
     return gpa > s.gpa;
   }

}
```

. . . .

A Lurking Problem

What if `Student` is already inheriting from another class?

```
class Person
```

```
{
   String name;
   String id;
}

class Student extends Person
{
   double gpa;
   String school;
}
```

. . . .

Why is This a Problem?

That would be no problem in C++, which permits multiple inheritance:

```
class Student: public Person, Comparable {
```

But Java only allows a class to have a single superclass, so we can't add `extends Comparable`

. . . .

A Java style Solution - Interface

Java programmers are more likely to approach this problem using an interface:

```
package java.lang;

public interface Comparable<T extends Object> {
   /**
       Compares this object with the specified object for order.
       Returns a negative integer, zero, or a positive integer
       as this object is less than, equal to, or greater than
       the specified object.
   */
   int compareTo (T other);
}
```

. . . .

Calling on a Comparable Interface

We use interfaces in our sorting routine just like we would use any kind of data type:

```java
public static void
    insertionSort (Comparable[] array)
  {
    for (int i = 1; i < array.length; ++i) {
        Comparable temp = array[i];
        int p = i;
        while ((p > 0)
          && temp.compareTo(array[p-1]) < 0) {
            array[p] = array[p-1];
            p--;
        }
        array[p] = temp;
    }
  }
```

. . . .

Implementing the Comparable Interface

But now our student class does not have to inherit

- It simply announces its intention to implement the interface:

```java
class Student extends Person
              implements Comparable<Student>
{
  double gpa;
  String school;
  int compareTo(Object o)
  {
    Student s = (Student)o;
    if (gpa < s.gpa)
      return -1;
    else if (gpa == s.gpa)
      return 0;
    else
      return 1;
  }
}
```

- Of course, having made that intention clear, we have to follow up by providing the required function(s) of the interface.
  - By the way, `Comparable` is not something I have made up - it's part of the Java API.

. . . .

This was Just an Example

- In practice, we would not write out own sorting function, but would use

```
package java.util;

public class Arrays
{
   ⋮
   public static void sort (Comparable[]);
   public static int binarySearch
       (Comparable[], Comparable key);
```

- and, to sort other data structures, the similar functions declared in `java.util.Collection`.

. . . .

21.3.3 Iterators in Java

java.util.Iterator

- A Java interface for looping through data structures

```
/*
 * Copyright (c) 1997, 2010, Oracle and/or its affiliates. All rights reserved.
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 *
 * This code is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 only, as
 * published by the Free Software Foundation.  Oracle designates this
 * particular file as subject to the "Classpath" exception as provided
 * by Oracle in the LICENSE file that accompanied this code.
 *
 * This code is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * version 2 for more details (a copy is included in the LICENSE file that
 * accompanied this code).
 *
 * You should have received a copy of the GNU General Public License version
 * 2 along with this work; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 */

package java.util;
```

```
/**
 * An iterator over a collection.  {@code Iterator} takes the place of
 * {@link Enumeration} in the Java Collections Framework.  Iterators
 * differ from enumerations in two ways:
 *
 * <ul>
 *      <li> Iterators allow the caller to remove elements from the
 *           underlying collection during the iteration with well-defined
 *           semantics.
 *      <li> Method names have been improved.
 * </ul>
 *
 * <p>This interface is a member of the
 * <a href="{@docRoot}/../technotes/guides/collections/index.html">
 * Java Collections Framework</a>.
 *
 * @param <E> the type of elements returned by this iterator
 *
 * @author  Josh Bloch
 * @see Collection
 * @see ListIterator
 * @see Iterable
 * @since 1.2
 */
public interface Iterator<E> {
    /**
     * Returns {@code true} if the iteration has more elements.
     * (In other words, returns {@code true} if {@link #next} would
     * return an element rather than throwing an exception.)
     *
     * @return {@code true} if the iteration has more elements
     */
    boolean hasNext();

    /**
     * Returns the next element in the iteration.
     *
     * @return the next element in the iteration
     * @throws NoSuchElementException if the iteration has no more elements
     */
    E next();

    /**
     * Removes from the underlying collection the last element returned
     * by this iterator (optional operation).  This method can be called
     * only once per call to {@link #next}.  The behavior of an iterator
     * is unspecified if the underlying collection is modified while the
     * iteration is in progress in any way other than by calling this
     * method.
     *
     * @throws UnsupportedOperationException if the {@code remove}
     *         operation is not supported by this iterator
     *
     * @throws IllegalStateException if the {@code next} method has not
     *         yet been called, or the {@code remove} method has already
     *         been called after the last call to the {@code next}
     *         method
     */
    void remove();
}
```

The key operations are:

- `hasNext()`: tests to see if there are more elements to be visited
- `next()`: advances the iterator to the next position and returns the element that it had been pointing to.
- Most containers provide one or more functions that return iterators.
  - The most common name for that function is `iterator()`

. . . .

Example: Using an Iterator

```
LinkedList<Book> books = new LinkedList<Book>();
books.add(cs330Text);
books.add(cs361Text);
books.add(cs252Text);
  ⋮
boolean found = false;
Iterator<Book> it = books.iterator();
while (it.hasNext() && !found)
{
  Book b = it.next();
  found = b.equals(cs252Text);
}
if (found)
{
  it.remove();
}
```

. . . .

Not as Strange as it Looks

Compare that to the C++ equivalent:

```
list<Book> books;
books.push_back(cs330Text);
books.push_back(cs361Text);
books.push_back(cs252Text);
  ⋮
bool found = false;
list<Book>::iterator it = books.begin();
while (it != books.end() && !found)
{
  found = (*it == cs252Text);
  ++it;
}
if (found)
{
  books.erase (it);
}
```

. . . .

Iterators: java and C++

Some rough correspondences:

Java	**C++**
`Iterator<E>`	`container<E>::iterator`
`container.iterator()`	`container.begin()`
`it.hasNext()`	`it != container.end()`
`x = it.next();`	`x = *it; it++;`
`it.remove();`	`container.erase(it);`

- Notice that you can't retrieve a value from the position denoted by a Java iterator without advancing the iterator.

. . . .

Limitations of the Java Iterator

- One consequence of that is that Java iterators are good for looping, but not useful as positions where something was found when searching. In C++ we might write:

  ```
  list<Book>::iterator find (list<Book>& books,
                             string title)
  {
    for (list<Book>::iterator it = books.begin();
          it != books.end(); ++it)
      {
       if (it->getTitle() == title)
          return it;
      }
    return books.end();
  }
  ```

  – There's no equivalent to that using Java iterators. Once we have used the iterator (to check the title), it no longer points at the same location, so we can't then return it as the position of the desired book.

. . . .

Limitations of the Java Iterator (cont.)

Similarly, there's no easy equivalent in Java to the C++ practice of using iterators as starting and ending positions of an operation:

```
template <typename Iterator>
Iterator copy (Iterator start, Iterator stop,
               Iterator dest)
{
   while (start != stop)
     {
       *dest = *start;
       ++start; ++dest;
```

```
    }
}
```

. . . .

Iterators and the “for each” Loop

Java *Iterator*s enable the simple “for each” loop syntax.

For any container type *C* that has a function `iterator()` that returns a value of type *Iterator<T>*, we can rewrite

```
C c = ...
  ⋮
for (Iterator<T> it = c.iterator();
        it.hasNext(); ) {
  T t = it.next();
    ⋮
}
```

by

```
C c = ...
  ⋮
for (T t: c) {
    ⋮
}
```

. . . .

“for-each” Loops and C++

- This for-each syntax is coming to C++
  - The C++11 standard provides the same syntax for containers that provide iterators via `begin()` and `end()`.

. . . .

Extending Interfaces

Interfaces can extend (inherit from) other interfaces.

```
/*
 * Copyright (c) 1997, 2007, Oracle and/or its affiliates. All rights reserved.
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 *
 * This code is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 only, as
 * published by the Free Software Foundation.  Oracle designates this
 * particular file as subject to the "Classpath" exception as provided
 * by Oracle in the LICENSE file that accompanied this code.
 *
 * This code is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * version 2 for more details (a copy is included in the LICENSE file that
 * accompanied this code).
 *
 * You should have received a copy of the GNU General Public License version
 * 2 along with this work; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 */

package java.util;

/**
```

```
* An iterator for lists that allows the programmer
* to traverse the list in either direction, modify
* the list during iteration, and obtain the iterator's
* current position in the list. A {@code ListIterator}
* has no current element; its <I>cursor position</I> always
* lies between the element that would be returned by a call
* to {@code previous()} and the element that would be
* returned by a call to {@code next()}.
* An iterator for a list of length {@code n} has {@code n+1} possible
* cursor positions, as illustrated by the carets ({@code ^}) below:
* <PRE>
*                      Element(0)   Element(1)   Element(2)   ... Element(n-1)
* cursor positions:  ^            ^            ^            ^                  ^
* </PRE>
* Note that the {@link #remove} and {@link #set(Object)} methods are
* <i>not</i> defined in terms of the cursor position;  they are defined to
* operate on the last element returned by a call to {@link #next} or
* {@link #previous()}.
*
* <p>This interface is a member of the
* <a href="{@docRoot}/../technotes/guides/collections/index.html">
* Java Collections Framework</a>.
*
* @author  Josh Bloch
* @see Collection
* @see List
* @see Iterator
* @see Enumeration
* @see List#listIterator()
* @since   1.2
*/
```

```
public interface ListIterator<E> extends Iterator<E> {
    // Query Operations

    /**
     * Returns {@code true} if this list iterator has more elements when
     * traversing the list in the forward direction. (In other words,
     * returns {@code true} if {@link #next} would return an element rather
     * than throwing an exception.)
     *
     * @return {@code true} if the list iterator has more elements when
     *         traversing the list in the forward direction
     */
    boolean hasNext();

    /**
     * Returns the next element in the list and advances the cursor position.
     * This method may be called repeatedly to iterate through the list,
     * or intermixed with calls to {@link #previous} to go back and forth.
     * (Note that alternating calls to {@code next} and {@code previous}
     * will return the same element repeatedly.)
     *
     * @return the next element in the list
     * @throws NoSuchElementException if the iteration has no next element
     */
    E next();

    /**
     * Returns {@code true} if this list iterator has more elements when
     * traversing the list in the reverse direction.  (In other words,
     * returns {@code true} if {@link #previous} would return an element
     * rather than throwing an exception.)
     *
     * @return {@code true} if the list iterator has more elements when
     *         traversing the list in the reverse direction
     */
    boolean hasPrevious();

    /**
     * Returns the previous element in the list and moves the cursor
     * position backwards.  This method may be called repeatedly to
     * iterate through the list backwards, or intermixed with calls to
     * {@link #next} to go back and forth.  (Note that alternating calls
     * to {@code next} and {@code previous} will return the same
     * element repeatedly.)
     *
     * @return the previous element in the list
     * @throws NoSuchElementException if the iteration has no previous
     *         element
     */
    E previous();

    /**
     * Returns the index of the element that would be returned by a
     * subsequent call to {@link #next}. (Returns list size if the list
     * iterator is at the end of the list.)
     *
     * @return the index of the element that would be returned by a
     *         subsequent call to {@code next}, or list size if the list
     *         iterator is at the end of the list
     */
    int nextIndex();
    /**
     * Returns the index of the element that would be returned by a
     * subsequent call to {@link #previous}. (Returns -1 if the list
     * iterator is at the beginning of the list.)
     *
     * @return the index of the element that would be returned by a
     *         subsequent call to {@code previous}, or -1 if the list
     *         iterator is at the beginning of the list
     */
    int previousIndex();


    // Modification Operations

    /**
     * Removes from the list the last element that was returned by {@link
     * #next} or {@link #previous} (optional operation).  This call can
     * only be made once per call to {@code next} or {@code previous}.
     * It can be made only if {@link #add} has not been
     * called after the last call to {@code next} or {@code previous}.
     *
     * @throws UnsupportedOperationException if the {@code remove}
     *         operation is not supported by this list iterator
     * @throws IllegalStateException if neither {@code next} nor
     *         {@code previous} have been called, or {@code remove} or
     *         {@code add} have been called after the last call to
     *         {@code next} or {@code previous}
     */
    void remove();

    /**
```

```
     * Replaces the last element returned by {@link #next} or
     * {@link #previous} with the specified element (optional operation).
     * This call can be made only if neither {@link #remove} nor {@link
     * #add} have been called after the last call to {@code next} or
     * {@code previous}.
     *
     * @param e the element with which to replace the last element returned by
     *          {@code next} or {@code previous}
     * @throws UnsupportedOperationException if the {@code set} operation
     *         is not supported by this list iterator
     * @throws ClassCastException if the class of the specified element
     *         prevents it from being added to this list
     * @throws IllegalArgumentException if some aspect of the specified
     *         element prevents it from being added to this list
     * @throws IllegalStateException if neither {@code next} nor
     *         {@code previous} have been called, or {@code remove} or
     *         {@code add} have been called after the last call to
     *         {@code next} or {@code previous}
     */
    void set(E e);

    /**
     * Inserts the specified element into the list (optional operation).
     * The element is inserted immediately before the element that
     * would be returned by {@link #next}, if any, and after the element
     * that would be returned by {@link #previous}, if any.  (If the
     * list contains no elements, the new element becomes the sole element
     * on the list.)  The new element is inserted before the implicit
     * cursor: a subsequent call to {@code next} would be unaffected, and a
     * subsequent call to {@code previous} would return the new element.
     * (This call increases by one the value that would be returned by a
     * call to {@code nextIndex} or {@code previousIndex}.)
     *
     * @param e the element to insert
     * @throws UnsupportedOperationException if the {@code add} method is
     *         not supported by this list iterator
     * @throws ClassCastException if the class of the specified element
     *         prevents it from being added to this list
     * @throws IllegalArgumentException if some aspect of this element
     *         prevents it from being added to this list
     */
    void add(E e);
}
```

- The *ListIterator* extends *Iterator* by adding a function to add data at a position:

```
LinkedList<Book> books = new LinkedList<Book>();
books.add(cs330Text);
books.add(cs361Text);
books.add(cs252Text);
  <:smvdots:>
boolean found = false;
ListIterator<Book> it = books.listIterator();
while (it.hasNext() && !found)
{
  Book b = (Book)it.next();
  found = (b.equals(cs252Text);
}
if (found)
{
  it.remove();
}
it.add(cs252textVersion2);
```

. . . .

Part VII

Applying OOP

Chapter 22

Functors

22.1 Functors in Java

Example: Sorting Reprise

- We want to provide a general-purpose sorting routine.
  - Previously, we used an interface to put a comparison function inside the classes to be sorted

. . . .

Things to Sort

```
class Student implements Comparable<Student>
{
   String name;
   double gpa;
     ⋮
   int compareTo(Student s)
```

```
    {
      if (gpa < s.gpa)
        return -1;
      else if (gpa == s.gpa)
        return 0;
      else
        return 1;
    }
  }
```

- What if, in the same program, we want to sort students by `name`?
  - We obviously can't have two functions `compareTo` in the same *Student* class.

. . . .

22.1.1 Functors

Functors

- A *functor* is an object that simulates a function.
- Provides an elegant solution to the dual-sorting-order problem
  - The sort routine would expect a functor to compare two objects
  - The application code would define functor classes that compare students by name and by gpa.

. . . .

A Comparable Functor Spec.

```
package java.util;
public interface Comparator<T extends object> {
  public int compare (T left, T right);
  public boolean equals (Object obj);
}
```

Not the same as *Comparable*:

```
public
interface Comparable<T extends Object> {
  public boolean comesBefore (T t);
}
```

- *Comparable* is something that a type *T* implements from so that an object of that type can compare *itself* to other objects.
- *Comparator* is something that you construct instances of so that *application code* can compare *pairs* of objects to each other.

. . . .

Comparator versus Comparable

- *Comparable* goes “inside” the class being compared
  - This means that *Comparable* is most often used for comparison activities that are "natural" or common to all uses of a class.
- *Comparator* is “outside” the class being compared
  - *Comparator*s are often more useful for comparison activities that arise from a specific *application* that uses the class.

. . . .

Sorting With Comparable

```
public static void
    insertionSort (Comparable[] array)
 {
   for (int i = 1; i < array.length; ++i) {
      Comparable temp = array[i];
      int p = i;
      while ((p > 0)
        && temp.compareTo(array[p-1]) < 0) {
         array[p] = array[p-1];
         p--;
      }
      array[p] = temp;
   }
 }
```

. . . .

Sorting With Functors

```
 public static <T> void insertionSort
   (T[] array, Comparator<T> compare)
 {
   for (int i = 1; i < array.length; ++i) {
      T temp = array[i];
      int p = i;
      while ((i > 0)
        && compare.compare(temp, array[p-1]) < 0) {
         array[p] = array[p-1];
         p--;
      }
```

```
        array[p] = temp;
    }
  }
```

. . . .

The Sorting Application

```
myListOfStudents = . . .;
Sorting.insertionSort(myListOfStudents,
                      numStudents,
                      compareByName);
printStudentDirectory (myListOfStudents);

Sorting.insertionSort(myListOfStudents,
                      numStudents,
                      compareByGPA);
printHonorsReport (myListOfStudents);
```

- Notice how `compareByName` and `compareByGPA` control the sorting order.

. . . .

compareByName

```
class CompareByName implements Comparator<Student>
{
  public int compare(Student left,
                     Student right)
  {
    return left.name.compareTo(right.name);
  }
}
```

. . . .

Compare By GPA

```
class CompareByGPA implements Comparator<Student>
{
  public int compare(Student left,
                     Student right)
  {
    if (left.gpa  < right.gpa)
      return -1;
    else if (left.gpa  == right.gpa)
      return 0;
    else
      return 1;
  }
}
```

. . . .

Back to the Application

```
myListOfStudents = ...;

CompareByName compareByName = new CompareByName();
Sorting.insertionSort(myListOfStudents,
     numStudents, compareByName);
printStudentDirectory (myListOfStudents);

CompareByGPA compareByGPA = new CompareByGPA();
Sorting.insertionSort(myListOfStudents,
     numStudents, compareByGPA);
printHonorsReport (myListOfStudents);
```

- The application simply creates the appropriate functor objects and then passes them to the sort function.

. . . .

Functors versus Members

- The Compare functors are not members of the Student class
  - do not affect its inheritance hierarchy
  - not really a part of the Student processing
    - but a part of the application algorithm

. . . .

22.1.2 Immediate Classes

Functors may be Throw-Away

- We don't really need those functors once we have passed them to the sort function
  - So actually inventing variables to hold them is a bit of a waste
  - Contributes to "namespace pollution",
    - falsely suggests to readers that they need to remember a name in case it gets used again later
    - Can also lead to conflicts later in the code if we mistakenly reuse the name for a different purpose

. . . .

Replacing Variables by Temporary Values

```
myListOfStudents = ...;

 Sorting.insertionSort(myListOfStudents,
                       numStudents,
                       new CompareByName());
 printStudentDirectory (myListOfStudents);
```

```
Sorting.insertionSort(myListOfStudents,
                      numStudents,
                      new CompareByGPA());
printHonorsReport (myListOfStudents);
```

. . . .

"Immediate" Classes in Java

Java even allows us to take that principle a step further.

- If we will never use a *class* after we have created a single instance of it, we can write the whole class declaration inside an algorithm.
- The syntax:

  ```
  new ClassName (params) {members}
  ```

  - declares a new, anonymous, subclass of *ClassName*, in which
    - certain *members*s are declared/overridden, and
  - allocates a single object of this new type

. . . .

Sorting With Immediate Classes

```
myListOfStudents = ...;
Sorting.insertionSort
  (myListOfStudents, numStudents,
    new Comparator<Student>() {
        public int compare(Student left,
                           Student right)
        {
```

```
            return left.name.compareTo(right.name);
          }
      });
printStudentDirectory (myListOfStudents);

Sorting.insertionSort
  (myListOfStudents, numStudents,
    new Comparator<Student>() {
        public int compare(Student left,
                           Student right)
        {
         if (left.gpa  < right.gpa)
           return -1;
         else if (left.gpa  == right.gpa)
           return 0;
         else
           return 1;
        }
      });
printHonorsReport (myListOfStudents);
```

....

Is This Really an Improvement?

- Immediate classes can make the code very hard to read.
- But can improve "locality" by keeping one-shot code blocks closer to the context in which they will actually be employed.

Most commonly used in GUI programming, where, as we will

- we commonly have many one-shot classes
  - one or more per button or control in a GUI screen

- and each such class provides one member function
  - which often has only a single line of code in its body

. . . .

22.2 Functors and GUIs

Functors and GUIs

Because functors are objects, they can be stored in data structures.

- Thus we can have data structures that collect "operations" to be performed at different times.
- This idea lies at the heart of the Java GUI model

. . . .

Observing GUI Elements

A complete GUI may contain many elements:

- windows
- menu bars and menus
- buttons, text entry boxes
- scroll bars, window resize controls, etc

. . . .

GUI Elements & Input Events

These elements may be targeted by various input *events*:

- mouse clicks, drags, etc.
- keys pressed
- element selected/activated

. . . .

pre-OO Event Handling

Input events in C/Windows programming:

```
while (event_available(process_id)) {
   e = next_event(process_id);
   switch (e.eventKind) {
     case MOUSECLICKED:
       if (e.target == MyButton1)
          b1Clicked();
       else if (e.target == MyButton2)
          b2Clicked();
       break;
     case WINDOWCLOSED:
       ⋮
```

. . . .

pre-OO Event Handling (cont.)

Unix/X let programmers register "callback functions" to be invoked when an event took place:

```
void b1Clicked(Event e) {. . .}
myButton1 = new_Button_Widget("Click Me");
register (myButton1, b1Clicked);
```

- No explicit event-handling loop was written by the programmer.
  - When input event occurs, X calls the registered callback for that event kind and GUI element

. . . .

OO Event Handling

The X model is more elegant and leads to much simpler code.

- But unsuitable for languages where functions can't be passed as parameters
- Also hard to pass additional, application-specific information to the callback function.

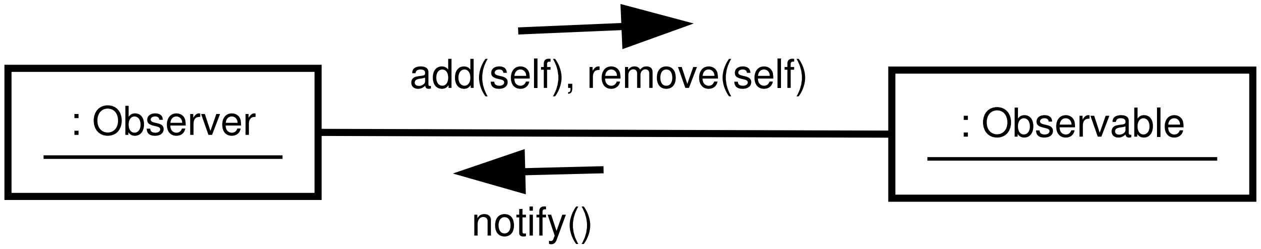
Functors overcome these limitations very nicely.

. . . .

22.2.1 Java Event Listeners

Java Event Listeners

- Each GUI element is subject to certain kinds of input events.
- Each GUI element keeps a list of Listeners to be notified when an input event occurs.
  - An instance of the Observer pattern



. . . .

Observing Events: Closing a Window

As an example of the java approach to listening for events, consider the problem of responding to a window being closed.

- This is the Java API for a class that listens for window events:

```
/*
 * @(#)WindowListener.java    1.6 96/12/17
 *
 * Copyright (c) 1995, 1996 Sun Microsystems, Inc. All Rights Reserved.
 *
 * This software is the confidential and proprietary information of Sun
 * Microsystems, Inc. ("Confidential Information").  You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
 * with Sun.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE
 * SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
 * PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY DAMAGES
 * SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING
 * THIS SOFTWARE OR ITS DERIVATIVES.
 *
 * CopyrightVersion 1.1_beta
 *
 */

package java.awt.event;

import java.util.EventListener;
```

```
/**
 * The listener interface for receiving window events.
 *
 * @version 1.6 12/17/96
 * @author Carl Quinn
 */
public interface WindowListener extends EventListener {
    /**
     * Invoked when a window has been opened.
     */
    public void windowOpened(WindowEvent e);

    /**
     * Invoked when a window is in the process of being closed.
     * The close operation can be overridden at this point.
     */
    public void windowClosing(WindowEvent e);

    /**
     * Invoked when a window has been closed.
     */
    public void windowClosed(WindowEvent e);

    /**
     * Invoked when a window is iconified.
     */
    public void windowIconified(WindowEvent e);

    /**
     * Invoked when a window is de-iconified.
     */
    public void windowDeiconified(WindowEvent e);

    /**
     * Invoked when a window is activated.
     */
    public void windowActivated(WindowEvent e);

    /**
     * Invoked when a window is de-activated.
     */
    public void windowDeactivated(WindowEvent e);
}
```

- An example of its use, declaring a listener and registering it as an observer of the window.

```
package SpreadSheetJ.ViewCon;

import SpreadSheetJ.Model.*;

import java.awt.*;
import java.awt.event.*;

import java.io.*;

// Thw window used to present the spreadsheet, including menu bar and
// formula bar.

public class MainWindow extends java.awt.Frame {

    private SSView ssview;
    private Formula formula;
    private TextField statusLine;
    private SpreadSheet sheet;
    private Clipboard clipboard;

    class WindowCloser implements WindowListener
    {
      public void windowClosing(WindowEvent e) {
        System.exit (0);
      }
      public void windowOpened(WindowEvent e) {}
      public void windowActivated(WindowEvent e) {}
        ⋮
    }


    public MainWindow (SpreadSheet s)
    {
        sheet = s;
        clipboard = new Clipboard();
          ⋮
        addWindowListener(new WindowCloser());

        setTitle ("SpreadSheet");
        buildMenu();
        pack();
        show();
    }
```

– The same, but using an immediate subclass

```
package SpreadSheetJ.ViewCon;
```

```
import SpreadSheetJ.Model.*;

import java.awt.*;
import java.awt.event.*;

import java.io.*;

// Thw window used to present the spreadsheet, including menu bar and
// formula bar.

public class MainWindow extends java.awt.Frame {

    private SSView ssview;
    private Formula formula;
    private TextField statusLine;
    private SpreadSheet sheet;
    private Clipboard clipboard;


    public MainWindow (SpreadSheet s)
    {
        sheet = s;
        clipboard = new Clipboard();
         ⋮

        addWindowListener(new WindowListener() {
           public void windowClosing(WindowEvent e) {
              System.exit (0);
             }
           public void windowOpened(WindowEvent e) {}
           public void windowActivated(WindowEvent e) {}
                ⋮
          });

        setTitle ("SpreadSheet");
        buildMenu();
        pack();
        show();
  }
```

....

Listeners & Adapters

Interfaces with multiple member functions are somewhat awkward here because

- you must implement ALL functions in the interface
- the interface cannot supply a default
- but for GUIs, a "do-nothing" default would be quite handy

"Adapter" Classes provide that default

....

WindowAdapter

- The WindowAdapter class

```
/*
 * @(#)WindowAdapter.java    1.7 97/01/03
 *
 * Copyright (c) 1995, 1996 Sun Microsystems, Inc. All Rights Reserved.
 *
 * This software is the confidential and proprietary information of Sun
 * Microsystems, Inc. ("Confidential Information").  You shall not
 * disclose such Confidential Information and shall use it only in
 * accordance with the terms of the license agreement you entered into
 * with Sun.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE
 * SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
 * PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY DAMAGES
 * SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING
 * THIS SOFTWARE OR ITS DERIVATIVES.
 *
 * CopyrightVersion 1.1_beta
 *
 */

package java.awt.event;

/**
 * The adapter which receives window events.
 * The methods in this class are empty;  this class is provided as a
 * convenience for easily creating listeners by extending this class
 * and overriding only the methods of interest.
 *
 * @version 1.7 01/03/97
 * @author Carl Quinn
 * @author Amy Fowler
 */
```

```
public abstract class WindowAdapter implements WindowListener {
    public void windowOpened(WindowEvent e) {}
    public void windowClosing(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
}
```

- An example of its use

```
package SpreadSheetJ.ViewCon;

import SpreadSheetJ.Model.*;

import java.awt.*;
import java.awt.event.*;

import java.io.*;

// Thw window used to present the spreadsheet, including menu bar and
// formula bar.

public class MainWindow extends java.awt.Frame {

    private SSView ssview;
    private Formula formula;
    private TextField statusLine;
    private SpreadSheet sheet;
    private Clipboard clipboard;
```

```
public MainWindow (SpreadSheet s)
{
    sheet = s;
    clipboard = new Clipboard();
   ⋮
    addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                    System.exit (0);
            }
        });

    setTitle ("SpreadSheet");
    buildMenu();
    pack();
    show();
}
```

– Not much different from what we had before, but we no longer need to explicitly provide empty operations for many events.

. . . .

Example: Building a Menu

As another example of listening for GUI events, consider setting up menus for an application.

- A Window may have a MenuBar
  - A MenuBar may have any number of Menus
    - Each Menu may have any number of MenuItems

. . . .

Menu Events

- MenuItems receive an ActionEvent when they are selected.

  -
    ```
    /*
     * @(#)ActionListener.java    1.6 96/11/23
     *
     * Copyright (c) 1995, 1996 Sun Microsystems, Inc. All Rights Reserved.
     *
     * This software is the confidential and proprietary information of Sun
     * Microsystems, Inc. ("Confidential Information").  You shall not
     * disclose such Confidential Information and shall use it only in
     * accordance with the terms of the license agreement you entered into
     * with Sun.
     *
     * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE
     * SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
     * IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
     * PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY DAMAGES
     * SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING
     * THIS SOFTWARE OR ITS DERIVATIVES.
     *
     * CopyrightVersion 1.1_beta
     *
     */

    package java.awt.event;

    import java.util.EventListener;
    ```

```
/**
 * The listener interface for receiving action events.
 *
 * @version 1.6 11/23/96
 * @author Carl Quinn
 */
public interface ActionListener extends EventListener {

    /**
     * Invoked when an action occurs.
     */
    public void actionPerformed(ActionEvent e);

}
```

- An example of building a menu

```
    private void buildMenu()
    {
    MenuBar menuBar = new MenuBar();     ❶

    Menu fileMenu = new Menu ("File");   ❷

    MenuItem loadItem = new MenuItem ("Load");   ❸
    fileMenu.add (loadItem);                     ❹
    loadItem.addActionListener(new ActionListener() {   ❺
     public void actionPerformed(ActionEvent e) {
         loadFile();
     }});
    if (inApplet == null) { // Applets can't write to the hard drive
        MenuItem saveItem = new MenuItem ("Save");
        fileMenu.add (saveItem);
        saveItem.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
     saveFile();
         }});
    }

    MenuItem exitItem = new MenuItem ("Exit");
    fileMenu.add (exitItem);
    exitItem.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
         if (inApplet != null) {
     hide();
     dispose();
         }
         else
     System.exit(0);
     }});


    Menu editMenu = new Menu ("Edit");           ❻

    MenuItem cutItem = new MenuItem ("Cut");
    editMenu.add (cutItem);
    cutItem.addActionListener(new ActionListener() {
     public void actionPerformed(ActionEvent e) {
         cutFormulae();
     }});
```

```
MenuItem copyItem = new MenuItem ("Copy");
editMenu.add (copyItem);
copyItem.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent e) {
     copyFormulae();
 }});

MenuItem pasteItem = new MenuItem ("Paste");
editMenu.add (pasteItem);
pasteItem.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent e) {
     pasteFormulae();
 }});


MenuItem eraseItem = new MenuItem ("Erase");
editMenu.add (eraseItem);
eraseItem.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent e) {
     eraseFormulae();
 }});



Menu helpMenu = new Menu ("Help");

MenuItem helpItem = new MenuItem ("Quick Help");
helpMenu.add (helpItem);
helpItem.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent e) {
     quickHelp();
```

```
    }});

   MenuItem aboutItem = new MenuItem ("About");
   helpMenu.add (aboutItem);
   aboutItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        about();
    }});


   menuBar.add (fileMenu);   ❼
   menuBar.add (editMenu);
   menuBar.add (helpMenu);

   setMenuBar (menuBar);    ❽
   }
```

❶ Create a menu bar

❷ Create a menu. Eventually we will have three menus in the bar: "File", "Edit", and "Help"

❸ Create a menu item

❹ Add it to the menu

❺ Register a listener that will call a function (`loadFile()`) when someone selects the menu item.

  * (No adapter required because ActionListener has only a single member function)

– Steps ❸. . . ❺ then get repeated over and over until we have all the items we want.

❻ Repeat ❷. . . ❺ for the Edit and Help menus

❼ Add all the menus to the menu bar

❽ Attach the menu bar to the window

. . . .

22.3 Functors in C++

Functors in C++

Do we need functors in a language where we can pass functions as parameters?

- Functors can do things normally reserved to objects
  - maintain state information
  - initialization & finalization

. . . .

Example: Selecting Students by GPA

- Old-style: passing functions directly

```
typedef bool *Selector(const Student&);

void listSelectedStudents(Student[] students,
                          int nStudents,
                          Selector selector)
{
  for (int i = 0; i < nStudents; ++i)
     if (selector(students[i]))
        cout << students[i] << endl;
}
```

. . . .

Application of listSelectedStudents

```
bool selectFailures (const Student& s)
{  return s.gpa < 1.0; }
```

```
bool selectHighAvg (const Student& s)
{  return s.gpa > 3.5; }

cout << "\fProbation List\n";
listSelectedStudents (allStudents, nStudents,
                      selectFailures);
cout << "\fDean's List\n";
listSelectedStudents (allStudents, nStudents,
                      selectHighAvg);
```

. . . .

Another Application

Suppose we want to be able to determine the range of desired GPA's dynamically:

```
cout << "Low GPA? " << flush;
cin >> lowGPA;
cout << "\nHigh GPA? " << flush;
cin >> highGPA;
listSelectedStudents (allStudents, nStudents,
                      selectRange);
```

- How can we write `selectRange`?

. . . .

Using a Function Style

- Make lowGPA and highGPA global variables:

```
double lowGPA, highGPA;

bool selectRange (const Student& s)
```

```
{
  return s.gpa >= lowGPA &&
         s.gpa <= highGPA;
}
```

- But I really *hate* global variables.

....

Using a Functor Style

```
class StudentSelectors {
public:
   bool test(const Student&) const =0;
};

void listSelectedStudents
      (Student[] students,
       int nStudents,
       StudentSelectors selector)
{
  for (int i = 0; i < nStudents; ++i)
     if (selector.test(students[i]))
        cout << students[i] << endl;
}
```

....

Application 1 with Functors

We can rewrite the first application with functors:

```
class SelectFailures: public StudentSelectors
{
   bool test (const Student& s) {return s.gpa < 1.0;}
};

class SelectHigh: public StudentSelectors
{
   bool test (const Student& s) {return s.gpa > 3.5; }
};

cout << "\fProbation List\n";
SelectFailures selectFailures;
listSelectedStudents (allStudents, nStudents,
                      selectFailures);
cout << "\fDean's List\n";
listSelectedStudents (allStudents, nStudents,
                      SelectHigh());
```

The application code itself is largely unchanged.

. . . .

Application 2 With Functors

The second application is cleaner (no globals) with functors:

```
class SelectRange: public StudentSelectors
{
  double lowGPA, highGPA;
public:
  SelectRange (double low, double high)
     : lowGPA(low), highGPA(high) { }

  bool test (const Student& s) const
  {
    return s.gpa >= lowGPA &&
            s.gpa <= highGPA;
  }
};
```

. . . .

Application 2 With Functors (cont.)

```
{
  double lowGPA, highGPA;
  cout << "Low GPA? " << flush;
  cin >> lowGPA;
  cout << "\nHigh GPA? " << flush;
  cin >> highGPA;
  listSelectedStudents
    (allStudents, nStudents,
     SelectRange(lowGPA, highGPA));
}
```

. . . .

22.3.1 operator()

Making Functors Pretty

- C++ does not have immediate subclasses as in Java
- But it does offer a special syntax designed to emphasize the function-like behavior or functor objects.

. . . .

What is f(x)?

Suppose you are reading some C++ code and encounter the expression: `f(x)`

- What is it?

- Obvious choice: `f` is a function and we are calling it with an actual parameter `x`.
- Less obvious: `f` could be a macro and we are calling it with an actual parameter `x`.
- OO idiom: `f` is a functor and we are calling its member function `operator()` with an actual parameter `x`.

. . . .

operator()

A special operator just for functors.

- Can be declared to take arbitrary parameters
- The shorthand for the call `x.operator()(y,z)` is `x(y,z)`.
  - Deliberately makes objects look like functions

. . . .

Student Listing Revisited

```
class StudentSelectors {
public:
   bool operator() (const Student&) const =0;
};

void listSelectedStudents
      (Student[] students,
       int nStudents,
       StudentSelectors selector)
{
  for (int i = 0; i < nStudents; ++i)
     if (selector(students[i]))
        cout << students[i] << endl;
}
```

. . . .

Application 1 with operator()

```
class SelectFailures: public StudentSelectors
{
   bool operator() (const Student& s)
      {return s.gpa < 1.0;}
};
class SelectHigh: public StudentSelectors
{
   bool operator() (const Student& s)
     {return s.gpa > 3.5; }
};

cout << "\fProbation List\n";
listSelectedStudents (allStudents, nStudents,
                      SelectFailures());
cout << "\fDean's List\n";
listSelectedStudents (allStudents, nStudents,
                      SelectHigh());
```

. . . .

SelectRange With operator()

```
class SelectRange: public StudentSelectors
{
  double lowGPA, highGPA;
public:
  SelectRange (double low, double high)
     : lowGPA(low), highGPA(high) { }

  bool operator() (const Student& s) const
  {
    return s.gpa >= lowGPA &&
           s.gpa <= highGPA;
  }
};
```

. . . .

Application 2 With operator()

```
{
  double lowGPA, highGPA;
  cout << "Low GPA? " << flush;
  cin >> lowGPA;
  cout << "\nHigh GPA? " << flush;
  cin >> highGPA;
  listSelectedStudents
    (allStudents, nStudents,
     SelectRange(lowGPA, highGPA));
}
```

. . . .

22.3.2 operator() and templates

operator() and templates

- operator() makes calls on functors look like calls on functions
- C++ templates allow us to write code in which type names are replaced by parameters to be filled in *at compile time*.
- This combination allows us to write code that can be used with either functors or "true" functions

. . . .

std Function Templates

- The C++ std library is packed with lots of small "algorithm fragments"
  - presented as function templates.

e.g.,

```
swap(x, y);
int smaller = min(0, 23);
string larger = max(string("Zeil"),
                    string("Adams"));
copy (v.begin(), v.end(), back_inserter(list));
```

....

std Templates & Functors

A number of the more interesting std template functions take function parameters. E.g.,

```
void printName (const Student& s)
{
  cout << s.name << endl;
}

for_each (students, students+nStudents,
          printName);
```

- Applies `printName` to each element in `students`.

....

for_each

Here's the code for `for_each`:

```
template <class I, class Function>
void for_each (I start, I stop, Function f) {
   for (; start != stop; ++start) {
      f(*start);
}
```

- Again, what is `f(x)`?
  - Could be a function or a functor

. . . .

Function and Functor are Interchangable

```
void printName (const Student& s)
{
  cout << s.name << endl;
}

class PrintGPA {
   void operator() (const Student& s) const
   { cout << s.gpa << endl; }
};

for_each (students, students+nStudents,
          printName); // function
for_each (students, students+nStudents,
          PrintGPA()); // functor
```

. . . .

Functors Add Flexibility

Functors often let us do things that simple functions would find awkward.

Hearkening back to our earlier example of the *SelectRange* functor:

```
Student* firstFailure =
   find_if(students, student+nStudents,
           SelectRange(0.0,0.999));
Student* firstPass =
   find_if(students, student+nStudents,
           SelectRange(1.0,4.0));
```

or

```
int numHonors = 0;
count(students, students+nStudents,
      SelectRange(3.5, 4.0), numHonors);
```

. . . .

Chapter 23

Graphic User Interfaces

The StringArt Program

To illustrate Java event handling, we'll look at a simple, but typical GUI interface to a graphic application

- `StringArt.java`
  - Compiled version of this program.

. . . .

The Process

1. Develop the model
2. Select the GUI elements to portray the model
3. Develop the layout of those elements
4. Add listeners to the elements

5. Implement custom drawing

. . . .

23.1 Develop the Model

Develop the Model

The *model* is the underlying data that is being manipulated and portrayed by the program.

- In this example, we need enough info to arrange the "strings" of various colors at appropriate steps around the circle.

```
// The Model
  private Color[] colors;
  private int stepSize = 5;
```

  - The array will hold two colors.

. . . .

Select the GUI elements to portray the model

- The core elements we will need:
  - a "canvas" on which to draw the graphics
    - This is generally done with a *JPanel*
  - A text entry box (*JTextField*) in which to enter the step size
  - Two buttons (*JButton*) to activate the color choices
- How do we choose these?
  - It helps to know what's available.

. . . .

23.2 Develop the layout of those elements

Develop the layout of those elements

There are two common approaches to laying out GUIs

- Manual arrangement using a program similar to a graphics editor (Visio, Dia, etc.) but with GUI elements as the building blocks
  - Easy & quick
  - Resulting layouts are often inflexible
    - Respond badly to window's being resized
    - Or to being displayed on smaller screens, different operating systems, window managers, etc.
- Programmed arrangement
  - Takes more effort
    - Though not much more given the need to program behaviors of the elements
  - Code can implement policies for resizing

. . . .

Programmed Approach in Java

Uses a combination of

- Container/nesting relationships
- Layout managers

. . . .

GUI Container Elements

- Some GUI elements are containers that hold other elements
  - The size and position of the container constrains the possible positions of the elements within it
  - Internal positioning is controlled by a layout manager.

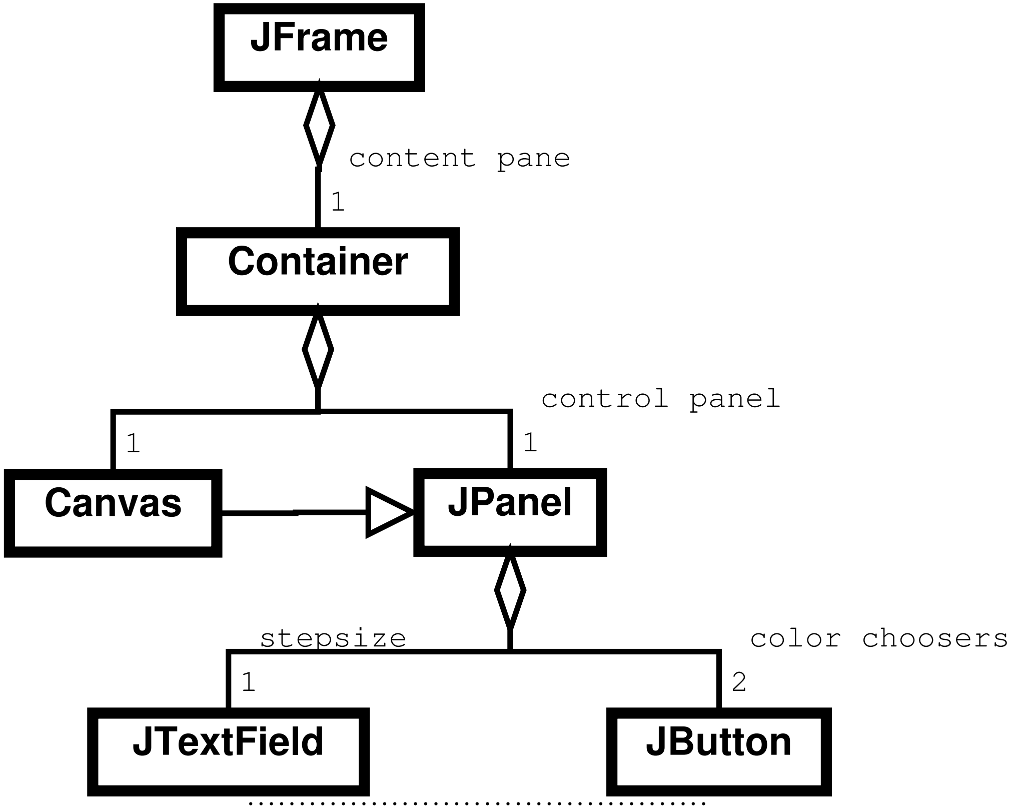
. . . .

GUI Elements in StringArt

- The outermost container is a window (*JFrame*)
  - This is a container.
  - Specifically, it contains a "content pane"
  - We add elements to the content pane
- The window's content pane will contain
  - Our drawing canvas (*JPanel*) and, beneath that,
  - A control panel (*JPanel*)
- The control panel will contain
  - The two buttons for choosing colors
  - the test extry box for the step size

. . . .

Structural Summary



Where Did the Control Panel Come From?

- Java has a handful of rules for easily arranging elements

  - horizontal rows
  - vertical columns
  - grids
  - arranged around a center, etc.

- It's easiest to break a desired arrangement down into a combination of these
  - e.g., horizontal row within the control panel, and vertical within the window

. . . .

Programming the Containment

GUI container elements are handled much like utility containers

```
public void createAndShowGUI() {
    window = new JFrame();      ❶
       ⋮
    canvas = new JPanel () {    ❷
        ⋮
        }
    };
  canvas.setBackground(Color.white);
  window.getContentPane().add (canvas, BorderLayout.CENTER);  ❸
  canvas.setPreferredSize(new Dimension(400, 400));

  JPanel controls = new JPanel();                             ❹

  colorChooser1 = new JButton("Color 1");                     ❺
  controls.add (colorChooser1);
    ⋮
  colorChooser2 = new JButton("Color 2");
```

```
    controls.add (colorChooser2);
      ⋮
    stepSizeIn = new JTextField (""+stepSize, 5);
    controls.add (stepSizeIn);
      ⋮
    window.getContentPane().add (controls, BorderLayout.SOUTH);  ❻
      ⋮
    window.pack();                                               ❼
    window.setVisible(true);
  }
```

❶ Create the window

❷ Create the canvas

❸ Add the canvas to the window's content pane

❹ Create the control panel

❺ Create the buttons and text field and add to the control panel

❻ Add the control panel to the window's content pane

❼ Arrange the components of the GUI and display the window.

. . . .

Laying out the components

Java has several layout managers.

- These give us several options for how to arrange our components
- Need to choose one for each container

- For the control panel, we want a horizontal row
  - *FlowLayout*, *BoxLayout*, or *BorderLayout* would work
  - *FlowLayout* is already the default for *JPanel*s, so it's easiest to stick with that
- For the window's content pane, we want a vertical row
  - *BoxLayout* or *BorderLayout* would work
  - I chose *BorderLayout*

. . . .

Using Border Layout

```
public void createAndShowGUI() {
    window = new JFrame();
    // set up the components
    window.getContentPane().setLayout (
          new BorderLayout());

    canvas = new JPanel () {
        ⋮
        }
    };
    ⋮
  window.getContentPane().add (canvas,
           BorderLayout.CENTER);

  JPanel controls = new JPanel();
    ⋮
  window.getContentPane().add (controls,
            BorderLayout.SOUTH);
    ⋮
```

. . . .

Miscellaneous Appearance Code

```
  public void createAndShowGUI() {
      window = new JFrame();
      // set up the components
      window.getContentPane().setLayout (new BorderLayout());

      canvas = new JPanel () {
          ⋮
          }
      };
    canvas.setBackground(Color.white);
    window.getContentPane().add (canvas, BorderLayout.CENTER);
canvas.setPreferredSize(new Dimension(400, 400));  ❶

    JPanel controls = new JPanel();

    colorChooser1 = new JButton("Color 1");
    controls.add (colorChooser1);
    setColor(colorChooser1, colors[0]);
    colorChooser1.addActionListener (new ColorChooser(colorChooser1, 0));

    colorChooser2 = new JButton("Color 2");
    controls.add (colorChooser2);
    setColor(colorChooser2, colors[1]);
    colorChooser2.addActionListener (new ColorChooser(colorChooser2, 1));

    stepSizeIn = new JTextField (""+stepSize, 5);
    controls.add (stepSizeIn);
    stepSizeIn.addActionListener (new ActionListener()
      {
        public void actionPerformed(ActionEvent e) {
          try {
            Integer newSize = new Integer(stepSizeIn.getText());
            stepSize = newSize.intValue();
            canvas.repaint();
          } catch (Exception ex) {};
        }
      });

    window.getContentPane().add (controls, BorderLayout.SOUTH);

    window.setDefaultCloseOperation((startedInAnApplet) ? JFrame.DISPOSE_ON_CLOSE : JFrame.EXIT_ON_CLOSE);

    window.pack();
    window.setVisible(true);
  }

  /**
   * Sets the background color of a button to the indicated color.
   * Changes the foreground to wither black or white, depending on
   * which will give more contrast agasint the new background.
   *
   * @param button
   * @param color
   */
  private void setColor(JButton button, Color color) {   ❷
    button.setBackground(color);
    int brightness = color.getRed() + color.getGreen() + color.getBlue(); // max of 3*255
    if (brightness > 3*255/2) {
        // This is a fairly bright color. Use black lettering
        button.setForeground (Color.black);
    } else {
        // This is a fairly dark color. Use white lettering
        button.setForeground (Color.white);
    }
}
```

Most of the highlighted code is self-explanatory

❶ The canvas, a panel with no internal compoenents, has no “natural” size and tends to shrink to 0×0 if we aren’t careful

❷ This utility function sets the background color of a button to match the color that it selects, and makes sure that the foreground color of the text is visible in contrast.

. . . .

23.3 Add listeners to the elements

Add listeners to the elements

We need to implement some behaviors

- Closing the window shuts down the program
- Entering a number in the text box changes the step size in the model
- Clicking a color button pops up a color selection dialog and updates the model

. . . .

Closing the Window

Here we use a shorthand technqiue rather than create a full-fledged functor to listen:

```
public void createAndShowGUI() {
    window = new JFrame();
        ⋮
  window.setDefaultCloseOperation(
    (startedInAnApplet)
        ? JFrame.DISPOSE_ON_CLOSE
        : JFrame.EXIT_ON_CLOSE);
        ⋮
}
```

. . . .

Entering the Step Size

This is handled by an *ActionListener*, which is triggered when we hit Enter/Return with the cursor positioned in the text box.

```
stepSizeIn = new JTextField (""+stepSize, 5);
controls.add (stepSizeIn);
stepSizeIn.addActionListener (new ActionListener()
  {
    public void actionPerformed(ActionEvent e) {
      try {
        Integer newSize = new Integer(
                       stepSizeIn.getText());
        stepSize = newSize.intValue();
             ⋮
      } catch (Exception ex) {};
    }
  });
```

. . . .

Color Selection

Buttons notify an *ActionListener* when a button is clicked

```
    colorChooser1 = new JButton("Color 1");
    controls.add (colorChooser1);
    setColor(colorChooser1, colors[0]);
    colorChooser1.addActionListener (
        new ColorChooser(colorChooser1, 0));

    colorChooser2 = new JButton("Color 2");
    controls.add (colorChooser2);
    setColor(colorChooser2, colors[1]);
    colorChooser2.addActionListener (
        new ColorChooser(colorChooser2, 1));
```

We implement this with our own subclass of *ActionListener*, *ColorChooser*.

. . . .

ColorChooser

```
  private class ColorChooser implements ActionListener {
      private JButton button;          ❶
      private int colorNum;

      public ColorChooser (JButton button, int colorNum) {
          this.button = button;        ❷
          this.colorNum = colorNum;
      }

      @Override
      public void actionPerformed(ActionEvent arg0) {   ❸
          Color chosen = JColorChooser.showDialog(window, "Choose a color",
                  colors[colorNum]);
          if (chosen != null) {
              colors[colorNum] = chosen;   ❹
```

```
                setColor (button, chosen);
                canvas.repaint();
            }
        }
    };
```

❶ Each *ColorChooser* object will remember which button it is attached to and which of the two colors it controls.

❷ That info is gathered and remembered in the constructor.

❸ Here is the function that we must provide if we are to implement *ActionListener*

The *JColorChooser* invoked in the first line pops up the stnadard Java color selection box.

❹ Once we have a selected color, we update the appropriate element of the model's *colors* array, and also change the button's coloring.

. . . .

23.4 Implement custom drawing

Implement custom drawing

The proper moment for drawing GUI elements is a bit tricky.
Think of some of the various situations in which all or part of a window needs to be redrawn:

- The program concludes some kind of computation and needs to display the results (i.e., a program state change).
- We had previously clicked on the minimize/iconify control to reduce the window to an icon, and now we click on the icon to open it back up to a full window.

. . . .

Implement custom drawing (cont.)

- The window was completley or partially hidden beneath other windows and we `Alt-tab` to bring it back up to the top.
- We drag another window or other item across the front of window, continuously hiding and revealing different potions of the window as we do so.

Any of these can force part or all of an application to need redrawing. Some of these (state changes) are under direct program control. Most are not.

. . . .

paint and repaint

- Most GUI elements already take care of drawing themselves.
- For elements with customized appearance:
  - Programmer provides a `paint` function.

```
    canvas = new JPanel () {
        public void paint (Graphics g) {
          super.paint(g);
          drawLines (g, getSize());
        }
      };
```

. . . .

Paint and Repaint

- We never call `paint` directly
- We call `repaint()`

  - That asks the system to schedule a call to `paint` for sometime in the near future.

- One reason for this indirect approach is that some actions (e.g., dragging things on the screen) can result in lots of repaint requests being sent within a very short period of time. If we actually had to redraw the GUI that often, machines with slower CPUs or graphics could be overwhelmed. The run-time system is allowed to "buffer" multiple requests to repaint the same GUI elements, so long as it does eventually call `paint()`.

. . . .

Repainting the StringArt Canvas

```
private class ColorChooser implements ActionListener {
    private JButton button;
    private int colorNum;

    public ColorChooser (JButton button, int colorNum) {
        this.button = button;
        this.colorNum = colorNum;
    }

    @Override
    public void actionPerformed(ActionEvent arg0) {
        Color chosen = JColorChooser.showDialog(window, "Choose a color", colors[colorNum]);
        if (chosen != null) {
            colors[colorNum] = chosen;
            setColor (button, chosen);
            canvas.repaint();
        }
    }
};
```

```
public void createAndShowGUI() {
    window = new JFrame();
    // set up the components
    window.getContentPane().setLayout (new BorderLayout());

    canvas = new JPanel () {
        public void paint (Graphics g) {
            super.paint(g);
            drawLines (g, getSize());
        }
    };
  canvas.setBackground(Color.white);
  window.getContentPane().add (canvas, BorderLayout.CENTER);
  canvas.setPreferredSize(new Dimension(400, 400));

  JPanel controls = new JPanel();

  colorChooser1 = new JButton("Color 1");
  controls.add (colorChooser1);
  setColor(colorChooser1, colors[0]);
  colorChooser1.addActionListener (new ColorChooser(colorChooser1, 0));

  colorChooser2 = new JButton("Color 2");
  controls.add (colorChooser2);
  setColor(colorChooser2, colors[1]);
  colorChooser2.addActionListener (new ColorChooser(colorChooser2, 1));

  stepSizeIn = new JTextField (""+stepSize, 5);
  controls.add (stepSizeIn);
  stepSizeIn.addActionListener (new ActionListener()
    {
      public void actionPerformed(ActionEvent e) {
        try {
          Integer newSize = new Integer(stepSizeIn.getText());
          stepSize = newSize.intValue();
          canvas.repaint();
        } catch (Exception ex) {};
      }
    });

  window.getContentPane().add (controls, BorderLayout.SOUTH);

  window.setDefaultCloseOperation((startedInAnApplet) ? JFrame.DISPOSE_ON_CLOSE : JFrame.EXIT_ON_CLOSE);

  window.pack();
  window.setVisible(true);
}
```

- We request a repaint of the canvas after any operation that could change its appearance.

. . . .

The Full Listing

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JColorChooser;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;

/**
 * A simple example of GUI event handling in a Java application.
 *
 * This can be run as a main program or as an applet.
 *
 * @author zeil
 *
 */


public class StringArt extends JApplet {

  private boolean startedInAnApplet;

// The Model
  private Color[] colors;
  private int stepSize = 5;

  // The View & Controls
  private JFrame window;
  private JPanel canvas;
  private JButton colorChooser1;
  private JButton colorChooser2;
  private JTextField stepSizeIn;


  private class ColorChooser implements ActionListener {
      private JButton button;
      private int colorNum;

      public ColorChooser (JButton button, int colorNum) {
          this.button = button;
          this.colorNum = colorNum;
      }

      @Override
      public void actionPerformed(ActionEvent arg0) {
          Color chosen = JColorChooser.showDialog(window, "Choose a color", colors[colorNum]);
          if (chosen != null) {
              colors[colorNum] = chosen;
              setColor (button, chosen);
              canvas.repaint();
          }
      }
  };

  public StringArt()
  {
      startedInAnApplet = false;
      window = null;
      colors = new Color[2];
      colors[0] = Color.red;
      colors[1] = Color.blue;
  }
```

```
public static void main (String[] args)
{
    StringArt instance = new StringArt();
    instance.createAndShowGUI();
}

public void createAndShowGUI() {
    window = new JFrame();
    // set up the components
    window.getContentPane().setLayout (new BorderLayout());

    canvas = new JPanel () {
        public void paint (Graphics g) {
            super.paint(g);
            drawLines (g, getSize());
        }
    };
  canvas.setBackground(Color.white);
  window.getContentPane().add (canvas, BorderLayout.CENTER);
  canvas.setPreferredSize(new Dimension(400, 400));

  JPanel controls = new JPanel();

  colorChooser1 = new JButton("Color 1");
  controls.add (colorChooser1);
  setColor(colorChooser1, colors[0]);
  colorChooser1.addActionListener (new ColorChooser(colorChooser1, 0));

  colorChooser2 = new JButton("Color 2");
```

```
    controls.add (colorChooser2);
    setColor(colorChooser2, colors[1]);
    colorChooser2.addActionListener (new ColorChooser(colorChooser2, 1));

    stepSizeIn = new JTextField (""+stepSize, 5);
    controls.add (stepSizeIn);
    stepSizeIn.addActionListener (new ActionListener()
      {
        public void actionPerformed(ActionEvent e) {
          try {
            Integer newSize = new Integer(stepSizeIn.getText());
            stepSize = newSize.intValue();
            canvas.repaint();
          } catch (Exception ex) {};
        }
      });

    window.getContentPane().add (controls, BorderLayout.SOUTH);

    window.setDefaultCloseOperation((startedInAnApplet) ? JFrame.DISPOSE_ON_CLOSE : JFrame.EXIT_ON_CLOSE);

    window.pack();
    window.setVisible(true);
  }

  /**
   * Sets the background color of a button to the indicated color.
   * Changes the foreground to wither black or white, depending on
   * which will give more contrast agasint the new background.
   *
   * @param button
   * @param color
   */
  private void setColor(JButton button, Color color) {
    button.setBackground(color);
    int brightness = color.getRed() + color.getGreen() + color.getBlue(); // max of 3*255
    if (brightness > 3*255/2) {
        // This is a fairly bright color. Use black lettering
        button.setForeground (Color.black);
    } else {
        // This is a fairly dark color. Use white lettering
        button.setForeground (Color.white);
    }
}

  // Applet functions

  public void init() {
      startedInAnApplet = true;
  }

public void start() {
    if (window == null)
        createAndShowGUI();
  }

  public void stop() {
  }

  public void destroy() {
  }
```

```
int interpolate (int x, int y, int i, int steps)
{
  return (i * x + (steps-i)*y) / steps;
}


Color interpolate(Color c1, Color c2, int i, int steps)
{
  return new Color (interpolate(c1.getRed(), c2.getRed(), i, steps),
                    interpolate(c1.getGreen(), c2.getGreen(), i, steps),
                    interpolate(c1.getBlue(), c2.getBlue(), i, steps));
}


class Point {
  double x;
  double y;
}

Point ptOnCircle (int degrees, int radius, Point center)
{
  Point p = new Point();
  double theta = Math.toRadians((double)degrees);
  p.x = center.x + (double)radius * Math.cos(theta);
  p.y = center.y + (double)radius * Math.sin(theta);
  return p;
}
```

```
public void drawLines(Graphics g, Dimension d)
{
  int dmin = (d.width < d.height) ? d.width : d.height;

  if (stepSize < 1)
    stepSize = 1;

  Point center = new Point();
  center.x = (double)d.width/2.0;
  center.y = (double)d.height/2.0;

  for (int i = 0; i < 60; ++i) {
    Point origin = ptOnCircle(6*i, dmin/2, center);
    int j = i + stepSize;
    while (j >= 60)
      j -= 60;
    while (i != j) {
      Point destination = ptOnCircle(6*j, dmin/2, center);
      g.setColor(interpolate(interpolate(colors[0], colors[1], i%30, 30),
                             interpolate(colors[0], colors[1], j, 60),
                             1, 2));
      g.drawLine ((int)origin.x, (int)origin.y,
                  (int)destination.x, (int)destination.y);
      j += stepSize;
      while (j >= 60)
        j -= 60;
    }
  }
}
```

```
}
```

. . . .

Chapter 24

Multi-Threading

Parallelism - Motivation

- Speed
- Responsiveness
- Clean design

. . . .

24.1 Overview

Concurrency

- Concurrency refers to the ability of a program to perform operations in parallel.
  - Faster on hardware with multiple CPU’s, or CPU’s with parallel capability

  - Cleaner designs (sometimes) even when only a single CPU available
  - Allows useful work when waiting for I/O

. . . .

24.1.1 Fundamental Ideas

Definitions

- Operations are *concurrent* if they could be executed in parallel.
  - Whether they are or not depends on hardware, compiler, etc.
- Operations that must occur in a fixed sequence are *sequential*.
- A *process* is a sequential calculation.
- Implemented in op sys as a separate execution state
  - execution counter & other registers
  - activation stack
  - possibly other data

. . . .

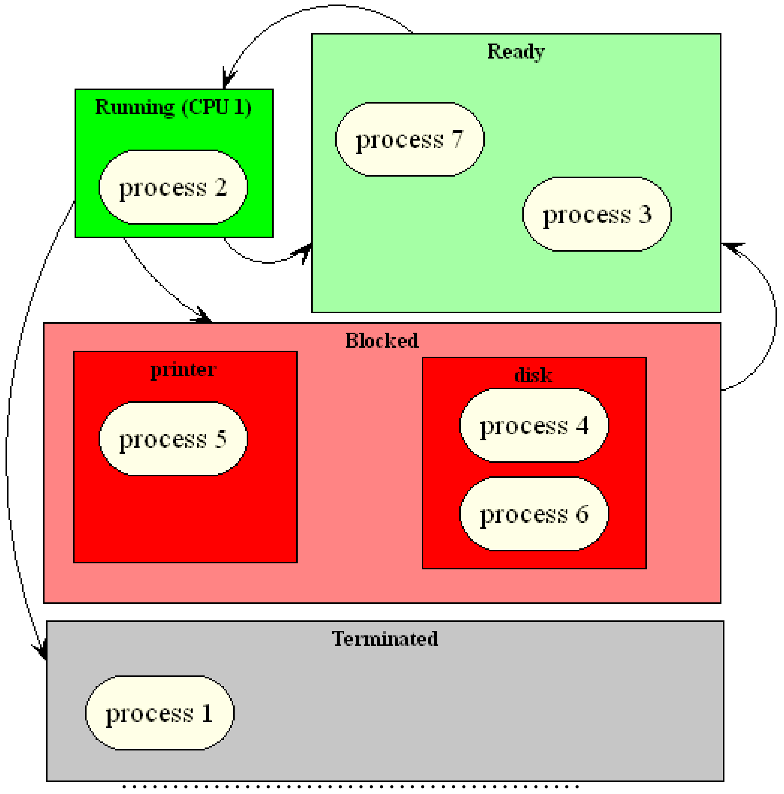
Scheduler

A *scheduler* is

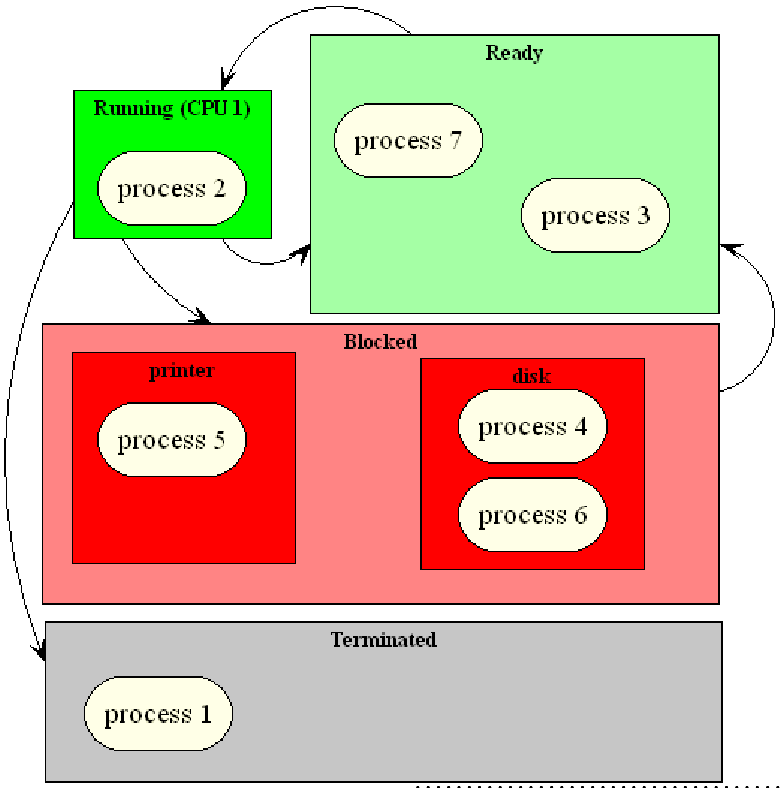
- responsible for deciding
  - which process gets the CPU,
  - and for how long
- Selection is made from among *ready* processes

. . . .

Process States



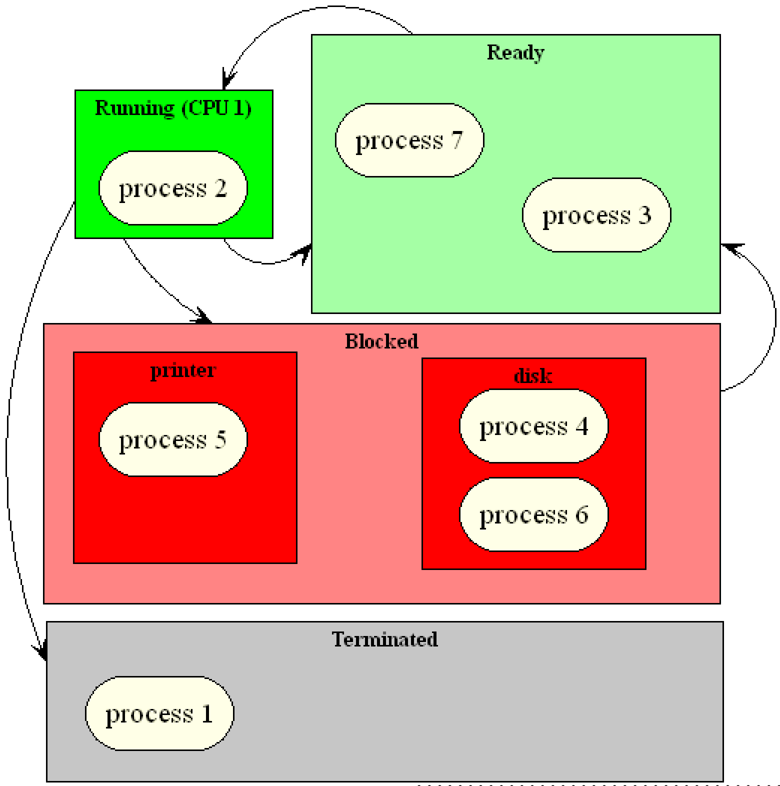
Running Processes



A process that is *running* may

- finish its computation
  - ⇒ `terminated`
- lose the CPU when its time slice runs out
  - ⇒ `ready`
- request a slow or unavailable resource
  - ⇒ `blocked`

Blocked Processes



A process that is *blocked* on some resource

- becomes `ready` when the resource is available
- still must wait for the scheduler to select it from among all the ready processes

Processes can Interact

Communication between process may be via

- messages
- shared variables

  - "shared" means visible to code of both processes

. . . .

Synchronization

Synchronization

- relates two or more threads
- restricts the order in which the collection of 2 or more processes can perform events
- Critical to making concurrent software safe

. . . .

24.1.2 Parallel versus Concurrent

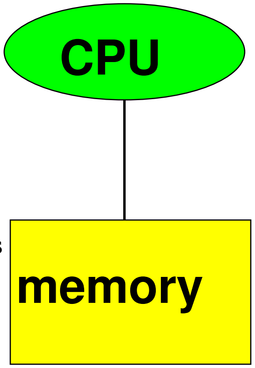
Parallel versus Concurrent

Concurrent software can run in many hardware configurations.

- single processor
- multi-processor
- distributed

. . . .

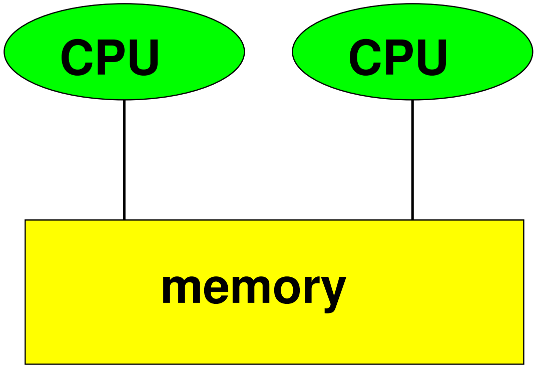
Single Processor



- one process can run while another is waiting for I/O
- time slicing
- reactive systems
  - e.g., user manipulates a windowed view of a lengthy calculation while that calculation is still in progress

. . . .

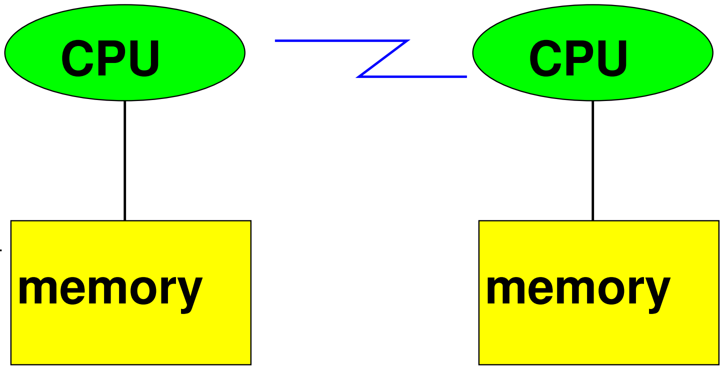
Multi-Processor



- CPU's may
  - work on distinct tasks,
  - or cooperate on a common goal
- Any single-CPU software designs can be mapped here

. . . .

Distributed Processors



- "shared" data must be replicated
- much harder to control
  - in many cases, the cheapest hardware solution
- e.g. off-the-shelf workstations connected via a network

Looking Ahead

- We'll concentrate on the shared memory models (single and multi-processors).

24.2 Spawning Processes

Spawning Processes

How do we get multiple processes in one program?

- Start with one, launch others
- May eventually need to assemble info from each spawned process into overall solution

24.2.1 Processes in Unix - Fork

Processes in Unix - Fork

- Usually, each time you type a command into the shell, a new process is launched to handle that command.
- Can also write programs that split into multiple processes

. . . .

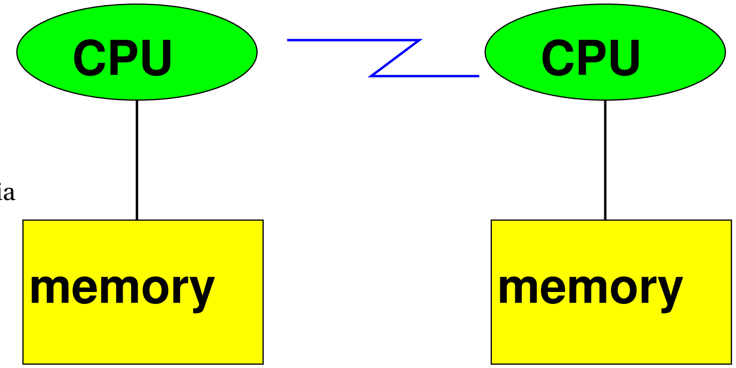
Concurrency in C++/Unix

Languages like C and C++ lack built-in support for concurrency.

- can sometimes do concurrent programming through special libraries
- tend to be OS and compiler-specific

. . . .

Unix Model



Unix employs a distributed process model.

- processes do not share memory but communicate via messages.

. . . .

Process Control

Basic operations:

- `fork()` creates a copy of the current process, identical down to every byte in memory, except that
  - the `fork()` call returns 0 to the new copy ("child") and
  - it returns the non-zero *process ID* of the child process to the original "parent"
- `wait(int*)` suspends the process until some child process completes.

. . . .

Example: Forking a Process

```
int status;
if (fork() == 0) {
   /* child 1 */
    execl("./prog1", "prog1", 0);
} else if (fork() == 0) {
   /* child 2 */
   execl("./prog2", "prog2", 0);
} else {
   /* parent */
   wait(&status);
   wait(&status);
}
```

. . . .

This program forks into two processes. The child process runs a program `prog1`, then finishes. The parent moves into the first `else`, then forks again into two processes. The second child runs a program `prog2`, then finishes. The parent moves to the final `else`, where it waits for the two children to finish.

Example: Web-Launched Grading

```
close STDOUT;
close STDERR;
$pid = fork();
if (!defined($pid)) {
     die ("Cannot fork: $!\n");
}

if ($pid) {
  #parent: issue web page to confirm that assignment has been received
     ⋮
} else {
  #child: launch a possibly lengthy autograde program
     $gradeResults = _autograde.pl -stop cleanup $grader $login_;
     $logFile = $assignmentDirectory . "grading.log";
     open (GRADELOG, ">>$logFile")
  }} die ("Could not append to $logFile: $!\n");
     print GRADELOG "$gradeResults\n";
     close GRADELOG;
}
```

. . . .

Here is part of the Perl script that gets launched when you submit programming assignments from the course's web-based submissions pages:

This is run as a CGI script when you click the submit button on the assignment submission pages. Now, the actual auto-grading may take several minutes, but web browsers will time out long before then. So the script forks to create a new process. The new child runs the auto-grader. In the meantime, the writes out a quick "Thank you for submitting" page that the browser can show immediately.

Process Communication

Unix processes can communicate via a *pipe*, a communication path that appears to the program as a pair of files, one for input, the other for output.

Reading from an empty pipe will block a process until data is available (written into the pipe by another process).

. . . .

Example of a Pipe

```
char str[6];
int mypipe[2];
int status;
pipe (mypipe); /* establishes a pipe.          */
               /* We read from mypipe[0] and   */
               /* write to mypipe[1]           */

if (fork() == 0) {      /* child */
   read(mypipe[0], str, 6);
   printf (str);
} else { /* parent */
   write (mypipe[1], "Hello", 6);
   wait (&status);
}
close mypipe[0];
close mypipe[1];
```

. . . .

This is "C" style I/O. Files are identified by an integer file handle, so out array of two ints is actually a pair of file handles. Each process can read from `mypipe[0]` and write to `mypipe[1]`, though in this particular example one process does all the writing and the other all the reading.

Overall

This approach to process communication is

- relatively simple
- relatively safe - lack of shared memory reduces the possibilities for processes to interfere with one another

  - still possible though (e.g., files)

- awkward for programs that need to pass complex data
- "heavy weight" - each process is relatively expensive

. . . .

Library vs. Language

Library-based concurrency tends to be non-portable.

- A big advantage to having concurrency built in as part of a language standard
  - But may lead to bad matches with some hardware configurations
- Only two such languages in common use: Ada and Java

. . . .

24.2.2 Heavy & Light Weight Processes

Heavy & Light Weight Processes

A Unix process includes its entire memory image

- Such *heavy-weight* processes may consume a lot of system resource and be slow to start up
- Discourages programs from spawning large numbers of child processes

. . . .

Threads

A *thread* is a *light-weight* process that

- Has its own execution location and activation stack
- Typically shares heap and static memory with its parent
- Some OS's support this directly
- Otherwise the Java runtime system implements multiple threads within a single heavy-weight OS process

. . . .

24.3 Java Threads

Java Threads

- Java has built-in support for threads
- In fact, Java programs with GUIs already have 2 or more threads
  - The *initial thread* starts in `main()`.
    - It usually builds the GUI and signals when it is to be displayed.
  - The *event handler* thread waits for input events and invokes the appropriate listeners when the events are detected.
    - `repaint()` calls are (usually?) handled as a simulated input event: queued up and eventually resulting in a `paint()`.

. . . .

Example: ThreadArt

- As an example of the use of threads in a Java GUI, let's add some dynamic behavior to the *StringArt* program of the earlier lesson on Java GUIs
- The thread will
  - Increment a counter that ranges from 0 to 99 and then back to zero
  - Schedule a `repaint()` for each value of the counter

. . . .

Using the counter to affect the drawing

```
public void drawLines(Graphics g, Dimension d)
{
    int dmin = (d.width < d.height) ? d.width : d.height;

    if (stepSize < 1)
        stepSize = 1;

    Point center = new Point();
    center.x = (double)d.width/2.0;
    center.y = (double)d.height/2.0;

    int k = Math.abs(cycleCounter - cycleLength/2);
    int theta = 60 * cycleCounter / cycleLength;

    for (int i = 0; i < 60; ++i) {
        int radius = dmin/2;
        Point origin = ptOnCircle(6*i+thetai, radius, center);
        int j = i + stepSize;
        while (j >= 60)
            j -= 60;
        while (i != j) {
            Point destination = ptOnCircle(6*j+theta, radius, center);
            Color c = iinterpolate(colors[0], colors[1], k, cycleLength/2);i
            g.setColor(c);
            g.drawLine ((int)origin.x, (int)origin.y,
                    (int)destination.x, (int)destination.y);
            j += stepSize;
            while (j >= 60)
                j -= 60;
        }
    }
}
```

- The counter is used to
  - Compute an angle offset
    - Which is added to the angles at which all points a computed
  - Interpolate between two colors

. . . .

24.3.1 Taking the Cheap Way Out

Taking the Cheap Way Out

The easiest way to animate this code is to

- Add a *Timer* to the program
  - Adds an action event to the queue every 50 milliseconds
- An action listener then can increment the counter

– and request a repaint

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JColorChooser;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.Timer;


/**
 * A simple example of GUI event handling in a Java application.
 *
 * This can be run as a main program or as an applet.
 *
 * @author zeil
 *
 */


public class ThreadArtByTimer extends JApplet {

    private boolean startedInAnApplet;
```

```
// The Model
private Color[] colors;
private int stepSize = 5;

private int cycleLength;
private int cycleCounter;

// The View & Controls
private JFrame window;
private JPanel canvas;
private JButton colorChooser1;
private JButton colorChooser2;
private JTextField stepSizeIn;

private Timer timer;


private class ColorChooser implements ActionListener {
    private JButton button;
    private int colorNum;

    public ColorChooser (JButton button, int colorNum) {
        this.button = button;
        this.colorNum = colorNum;
    }

    @Override
    public void actionPerformed(ActionEvent arg0) {
        Color chosen = JColorChooser.showDialog(window, "Choose a color", colors[colorNum]);
        if (chosen != null) {
            colors[colorNum] = chosen;
            setColor (button, chosen);
            canvas.repaint();
        }
    }
};


/**
 * Action that slowly changes the color of the drawing
 *
 */
public class ColorChanger implements ActionListener {

    public ColorChanger()
    {
    }



    @Override
    public void actionPerformed(ActionEvent arg0) {
        cycleCounter = (cycleCounter + 1) % cycleLength;
        canvas.repaint();
    }

}
public ThreadArtByTimer()
{
    startedInAnApplet = false;
    window = null;
    colors = new Color[2];
    colors[0] = Color.red;
    colors[1] = Color.blue;
    cycleLength = 100;
    cycleCounter = 0;
}


public static void main (String[] args)
{
    ThreadArtByTimer instance = new ThreadArtByTimer();
    instance.createAndShowGUI();
}

public void createAndShowGUI() {
    window = new JFrame();
    // set up the components
    window.getContentPane().setLayout (new BorderLayout());

    canvas = new JPanel () {
        public void paint (Graphics g) {
            super.paint(g);
            drawLines (g, getSize());
        }
    };
    canvas.setBackground(Color.white);
    window.getContentPane().add (canvas, BorderLayout.CENTER);
    canvas.setPreferredSize(new Dimension(400, 400));

    JPanel controls = new JPanel();

    colorChooser1 = new JButton("Color 1");
    controls.add (colorChooser1);
    setColor(colorChooser1, colors[0]);
    colorChooser1.addActionListener (new ColorChooser(colorChooser1, 0));

    colorChooser2 = new JButton("Color 2");
    controls.add (colorChooser2);
    setColor(colorChooser2, colors[1]);
    colorChooser2.addActionListener (new ColorChooser(colorChooser2, 1));

    stepSizeIn = new JTextField (""+stepSize, 5);
    controls.add (stepSizeIn);
    stepSizeIn.addActionListener (new ActionListener()
    {
        public void actionPerformed(ActionEvent e) {
            try {
                Integer newSize = new Integer(stepSizeIn.getText());
                stepSize = newSize.intValue();
                canvas.repaint();
            } catch (Exception ex) {};
        }
    });

    window.getContentPane().add (controls, BorderLayout.SOUTH);

    window.setDefaultCloseOperation((startedInAnApplet) ? JFrame.DISPOSE_ON_CLOSE : JFrame.EXIT_ON_CLO
    timer = new Timer(50, new ColorChanger());
    timer.start();

    window.pack();
    window.setVisible(true);
}

/**
 * Sets the background color of a button to the indicated color.
 * Changes the foreground to wither black or white, depending on
 * which will give more contrast agasint the new background.
 *
 * @param button
 * @param color
 */
private void setColor(JButton button, Color color) {
    button.setBackground(color);
    int brightness = color.getRed() + color.getGreen() + color.getBlue(); // max of 3*255
    if (brightness > 3*255/2) {
        // This is a fairly bright color. Use black lettering
        button.setForeground (Color.black);
    } else {
        // This is a fairly dark color. Use white lettering
        button.setForeground (Color.white);
    }
}

// Applet functions

public void init() {
    startedInAnApplet = true;
}

public void start() {
    if (window == null)
        createAndShowGUI();
}

public void stop() {
}

public void destroy() {
}




int interpolate (int x, int y, int i, int steps)
{
    return (i * x + (steps-i)*y) / steps;
}


Color interpolate(Color c1, Color c2, int i, int steps)
{
    return new Color (interpolate(c1.getRed(), c2.getRed(), i, steps),
            interpolate(c1.getGreen(), c2.getGreen(), i, steps),
            interpolate(c1.getBlue(), c2.getBlue(), i, steps));
}
```

```
class Point {
    double x;
    double y;
}

Point ptOnCircle (int degrees, int radius, Point center)
{
    Point p = new Point();
    double theta = Math.toRadians((double)degrees);
    p.x = center.x + (double)radius * Math.cos(theta);
    p.y = center.y + (double)radius * Math.sin(theta);
    return p;
}

public void drawLines(Graphics g, Dimension d)
{
    int dmin = (d.width < d.height) ? d.width : d.height;

    if (stepSize < 1)
        stepSize = 1;

    Point center = new Point();
    center.x = (double)d.width/2.0;
    center.y = (double)d.height/2.0;

    int k = Math.abs(cycleCounter - cycleLength/2);
    int theta = 60 * cycleCounter / cycleLength;

    for (int i = 0; i < 60; ++i) {
        int radius = dmin/2; //interpolate(dmin/4, dmin/2, k, cycleLength/2);
        Point origin = ptOnCircle(6*i+theta, radius, center);
```

```
            int j = i + stepSize;
            while (j >= 60)
                j -= 60;
            while (i != j) {
                Point destination = ptOnCircle(6*j+theta, radius, center);
                Color c = interpolate(colors[0], colors[1], k, cycleLength/2);
                g.setColor(c);
                g.drawLine ((int)origin.x, (int)origin.y,
                        (int)destination.x, (int)destination.y);
                j += stepSize;
                while (j >= 60)
                    j -= 60;
            }
        }
    }


}
```

. . . .

Let's Pretend That Never Happened

- Timers are easy to work with, but not always powerful enough
  - Not good if the recurring event may sometimes take a long time
    - Because it runs in the event handler thread, a time-consuming listener would "lock up" the GUI
  - Does not offer very flexible control options
- For educational purposes, therefore, let's look at how we might move that timed action into a separate thread
  - *not* the event handler thread

. . . .

24.3.2 Working with Threads

The Java Thread class

- Any subclasses of the system class *Thread* can have a function member `run()` that executes as a distinct process (per object).
  - *Never* call `run()` directory.
- A *Thread* object is not actually a new thread (process) unless we `start()` it
  - The `start` function
    * creates a new thread with its own activation stack, but sharing heap memory with the other threads
    * Starts that thread runnign with a call to the object's `run()` function.
  - The thread terminates when control returns from `run()`.

. . . .

ThreadArt

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JColorChooser;
import javax.swing.JFrame;
import javax.swing.JPanel;
```

```java
import javax.swing.JTextField;


/**
 * A simple example of GUI event handling in a Java application.
 *
 * This can be run as a main program or as an applet.
 *
 * @author zeil
 *
 */


public class ThreadArt extends JApplet {

    private boolean startedInAnApplet;

    // The Model
    private Color[] colors;
    private int stepSize = 5;

    private int cycleLength;
    private int cycleCounter;
    private boolean running;

    // The View & Controls
    private JFrame window;
    private JPanel canvas;
    private JButton colorChooser1;
    private JButton colorChooser2;
    private JTextField stepSizeIn;
    private Animator colorChanger;


    private class ColorChooser implements ActionListener {
        private JButton button;
        private int colorNum;

        public ColorChooser (JButton button, int colorNum) {
            this.button = button;
            this.colorNum = colorNum;
        }

        @Override
        public void actionPerformed(ActionEvent arg0) {
            Color chosen = JColorChooser.showDialog(window, "Choose a color", colors[colorNum]);
            if (chosen != null) {
                colors[colorNum] = chosen;
                setColor (button, chosen);
                canvas.repaint();
            }
        }
    };


    /**
     * Thread that slowly changes the color of the drawing
     *
     */
    public class Animator extends Thread {   ❶
```

```
        public Animator()
        {
        }


        public void run()                    ❷
        {
            running = true;
            while (running) {                ❸
                try {
                    sleep(50);               ❹
                } catch (InterruptedException e) {
                  break;
                }
                cycleCounter = (cycleCounter + 1) % cycleLength; ❺
                canvas.repaint();
            }
        }

    }




    public ThreadArt()
    {
        startedInAnApplet = false;
        window = null;
        colors = new Color[2];
        colors[0] = Color.red;
        colors[1] = Color.blue;
        cycleLength = 100;
        cycleCounter = 0;
        running = false;
    }


    public static void main (String[] args)
    {
        ThreadArt instance = new ThreadArt();
        instance.createAndShowGUI();
    }

    public void createAndShowGUI() {
        window = new JFrame();
        // set up the components
        window.getContentPane().setLayout (new BorderLayout());

        canvas = new JPanel () {
            public void paint (Graphics g) {
                super.paint(g);
                drawLines (g, getSize());
            }
        };
        canvas.setBackground(Color.white);
        window.getContentPane().add (canvas, BorderLayout.CENTER);
        canvas.setPreferredSize(new Dimension(400, 400));

        JPanel controls = new JPanel();

        colorChooser1 = new JButton("Color 1");
        controls.add (colorChooser1);
        setColor(colorChooser1, colors[0]);
        colorChooser1.addActionListener (new ColorChooser(colorChooser1, 0));

        colorChooser2 = new JButton("Color 2");
        controls.add (colorChooser2);
        setColor(colorChooser2, colors[1]);
        colorChooser2.addActionListener (new ColorChooser(colorChooser2, 1));

        stepSizeIn = new JTextField (""+stepSize, 5);
        controls.add (stepSizeIn);
        stepSizeIn.addActionListener (new ActionListener()
        {
            public void actionPerformed(ActionEvent e) {
                try {
                    Integer newSize = new Integer(stepSizeIn.getText());
                    stepSize = newSize.intValue();
                    canvas.repaint();
                } catch (Exception ex) {};
            }
        });

        window.getContentPane().add (controls, BorderLayout.SOUTH);

        window.setDefaultCloseOperation((startedInAnApplet) ? JFrame.DISPOSE_ON_CLOSE : JFrame.EXIT_ON_CLO


        colorChanger = new Animator();   ❻
        colorChanger.start();

        window.pack();
        window.setVisible(true);
```

```
    }

    /**
     * Sets the background color of a button to the indicated color.
     * Changes the foreground to wither black or white, depending on
     * which will give more contrast agasint the new background.
     *
     * @param button
     * @param color
     */
    private void setColor(JButton button, Color color) {
        button.setBackground(color);
        int brightness = color.getRed() + color.getGreen() + color.getBlue(); // max of 3*255
        if (brightness > 3*255/2) {
            // This is a fairly bright color. Use black lettering
            button.setForeground (Color.black);
        } else {
            // This is a fairly dark color. Use white lettering
            button.setForeground (Color.white);
        }
    }

    // Applet functions

    public void init() {
        startedInAnApplet = true;
    }

    public void start() {
        if (window == null)
            createAndShowGUI();
    }

    public void stop() {
    }

    public void destroy() {
    }





    int interpolate (int x, int y, int i, int steps)
    {
        return (i * x + (steps-i)*y) / steps;
    }


    Color interpolate(Color c1, Color c2, int i, int steps)
    {
        return new Color (interpolate(c1.getRed(), c2.getRed(), i, steps),
                interpolate(c1.getGreen(), c2.getGreen(), i, steps),
                interpolate(c1.getBlue(), c2.getBlue(), i, steps));
    }


    class Point {
        double x;
        double y;
    }

    Point ptOnCircle (int degrees, int radius, Point center)
    {
        Point p = new Point();
        double theta = Math.toRadians((double)degrees);
        p.x = center.x + (double)radius * Math.cos(theta);
        p.y = center.y + (double)radius * Math.sin(theta);
        return p;
    }

    public void drawLines(Graphics g, Dimension d)
    {
        int dmin = (d.width < d.height) ? d.width : d.height;

        if (stepSize < 1)
            stepSize = 1;

        Point center = new Point();
        center.x = (double)d.width/2.0;
        center.y = (double)d.height/2.0;

        int k = Math.abs(cycleCounter - cycleLength/2);
        int theta = 60 * cycleCounter / cycleLength;

        for (int i = 0; i < 60; ++i) {
            int radius = dmin/2;
            Point origin = ptOnCircle(6*i+theta, radius, center);
            int j = i + stepSize;
            while (j >= 60)
                j -= 60;
            while (i != j) {
                Point destination = ptOnCircle(6*j+theta, radius, center);
                Color c = interpolate(colors[0], colors[1], k, cycleLength/2);
                g.setColor(c);
                g.drawLine ((int)origin.x, (int)origin.y,
                        (int)destination.x, (int)destination.y);
                j += stepSize;
                while (j >= 60)
                    j -= 60;
            }
        }
    }


}
```

❶ Here we declare a *Thread* subclass

❷ . . . and override the `run()` function to manage our counter

❸ Loop almost forever

❹ The `sleep` function blocks the thread for at the indicated number of milliseconds

❺ This is what we came here to do.

The repaint() will cause a near-future redraw of the canvas, which will use our new value of *cycleCounter*

❻ Once the GUI has been set up, we create the new thread object and `start()` it.

. . . .

24.4 Interleaving & Synchronization

Interleaving & Synchronization

- In what order do calculations take place?

- How Does Order Affect Correctness?

. . . .

Interleavings

Given 2 processes whose code looks like:

process 1	**process 2**
a;	x;
b;	y;
c;	z;

. . . .

Possible Orderings

First statement must be either a or x.

- If it's a, then the 2nd statement executed must be either b or x
- If it's x, then the 2nd statement executed must be either a or y

. . . .

Possible Orderings (cont.)

Expanding that:
First statement must be either a or x.

- If it's a, then the 2nd statement executed must be either b or x
  - If it's b, then the 3rd must be either c or x.
  - If it's x, then the 3rd must be either b or y.
- If it's x, then the 2nd statement executed must be either a or y

. . . .

and so on

Possible Orderings

`abcxyz, abxcyz, abxycz, abxyzc, axbcyz, axbycz, axbyzc, axybcz, axybzc, axyzbc, xabcyz, xabycz, xabyzc, xaybcz, xaybzc, xayzbc, xyabcz, xyabzc, xyazbc, xyzabc`

Note that, although there are many possibilities, b never precedes a, c never precedes a or b, etc.

. . . .

Interleavings

An *interleaving* of two sequences `s` and `t` is any sequence `u`

- formed from the events of `s` and `t` such that
  - the events of `s` retain their position relative to one another and
  - the events of `t` do likewise.

. . . .

Describing Process Execution

- Describe each process as a sequence of "atomic" actions.
  - What constitutes "atomic" depends on hardware, compiler, etc.
- Each interleave of the two processes is a possible execution order for the program as a whole.
  - (assuming no synchronization takes place)

. . . .

24.4.1 Safety and Parallel Programs

Shared Resources

Processes often compete for shared resources.

- I/O devices
- shared variables
- files

In these cases, some interleaves are dangerous.

. . . .

Safety

We say that a nondeterministic program is *safe* if all of the answers it provides for any input are acceptable.

. . . .

Example

Processes with Shared Variable

process 1	**process 2**
`x = x + 1;`	`x = x + 1;`

Assuming that x starts with a value of -1 before the two processes are started, what will the value of x be after both have finished?

. . . .

We could get: 1

Processes with Shared Variable

process 1	**process 2**
`x = x + 1;`	`x = x + 1;`

- process 1 fetches x (-1)

- process 1 adds 1 to x and stores it (x == 0)
- process 2 fetches x (0)
- process 2 adds 1 to x and stores it (x == 1)

....

We could get: 0

Processes with Shared Variable

process 1	**process 2**
`x = x + 1;`	`x = x + 1;`

- process 1 fetches x (-1)
- process 2 fetches x (-1)
- process 1 adds 1 to (it's copy of) x and stores it (x == 0)
- process 2 adds 1 to (it's copy of) x and stores it (x == 0)

....

Nondeterminism

- Now these two alternatives might or might not be bad.
- Some concurrent programs are specifically designed for a limited amount of *nondeterminism*:
  - The property of having multiple possible answers for the same input
- But there are other possibilities as well.

....

We could get: 65536

Processes with Shared Variable

process 1	process 2
`x = x + 1;`	`x = x + 1;`

- process 1 fetches x (-1)
- process 1 adds 1 to (it's copy of) x
- process 1 begins storing x. Stores the higher two bytes (x == 0000FFFFH)
- process 2 fetches x (65535)
- process 1 stores remaining 2 bytes of x (`x == 0`)
- process 2 adds 1 to (it's copy of) xand stores it (x== 65536)

. . . .

65536 : Really?

- Admittedly, this requires a different level of atomicity than we might have expected, but this is not at all impossible in a distributed system.
- And it's a pretty good bet that we would not consider this an acceptable output of these two processes.
  - Which implies that this code is *unsafe*.

. . . .

Simultaneous Access to Compound Data

More realistically, any kind of compound data structure is likely to be sensitive to simultaneous update by different processes.

More specifically,

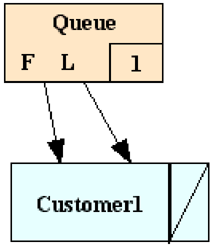
- Simultaneous read-only access is probably safe.
- Simultaneous updates are likely to be a problem.
- Simultaneous reading and updating are often a problem as well.

. . . .

Example: Simultaneous Access to a Queue

Suppose that we have a queue of customer records,

- implemented as a linked list
- currently holding one record



```
add(data)
L.next = new Node(data);
L = L.next
++count
```

```
remove()
tmp = F,next;
F.next = null
F = tmp
--count
```

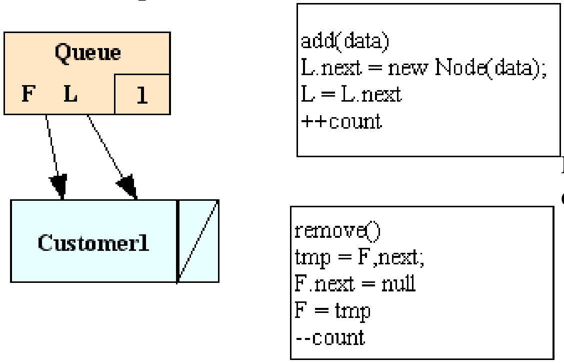
Here we show the data and possible implementations of the functions to add to the end and remove from the front.

- Verify for yourself that, if either function runs without interruption, this code would work
  - No matter which of the two runs first.

. . . .

Is there a potential problem with simultaneous access to these structures? The sequence of pictures below show that, for a linked list implementation of the queue, we could indeed get into trouble if a queue has a single element in it and one thread then tries to add a new element while another thread tries to remove one. (There may be other scenarios in which simultaneous access would also fail.)

Queues Interrupted

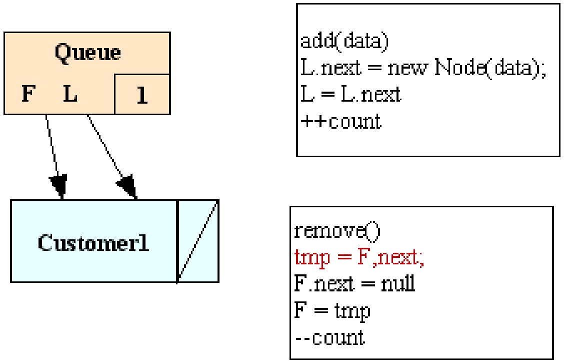


```
add(data)
L.next = new Node(data);
L = L.next
++count
```

```
remove()
tmp = F,next;
F.next = null
F = tmp
--count
```

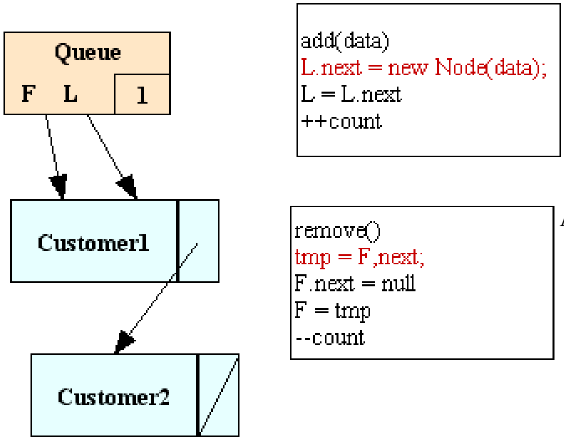
Now suppose that one process tries to simultaneously add a record to the end, and another one tries to remove a record from the front.

Assume that the `remove()` call gets the CPU first.



```
add(data)
L.next = new Node(data);
L = L.next
++count
```

```
remove()
tmp = F,next;
F.next = null
F = tmp
--count
```

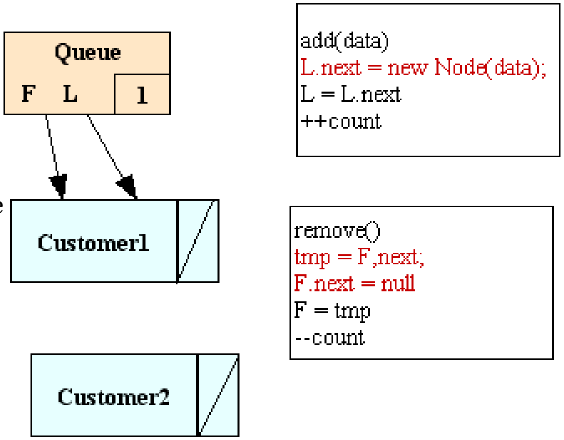


```
add(data)
L.next = new Node(data);
L = L.next
++count
```

```
remove()
tmp = F.next;
F.next = null
F = tmp
--count
```

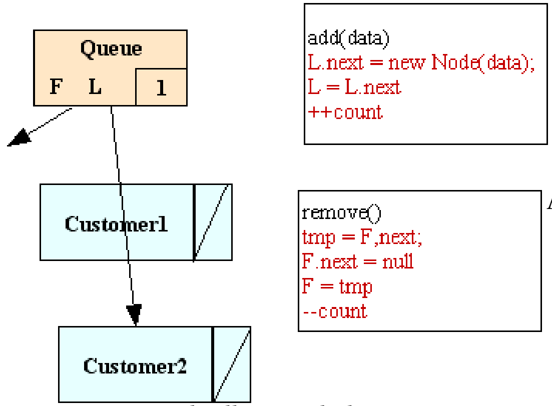
And then the `add()` gets to run for a little while. . .

and then control switches back to the `remove()` call, which runs to the end.



```
add(data)
L.next = new Node(data);
L = L.next
++count
```

```
remove()
tmp = F.next;
F.next = null
F = tmp
--count
```



```
add(data)
L.next = new Node(data);
L = L.next
++count
```

```
remove()
tmp = F,next;
F.next = null
F = tmp
--count
```

And finally the `add()` is allowed to complete.

Our queue is now badly mangled.

. . . .

24.4.2 Synchronization

Synchronization

Synchronization is a restriction of the possible interleavings of a set of processes.

- (to protect against interleaves that produce undesirable results.)

. . . .

Synch & Shared Resources

Synchronization often takes the form of protecting a shared resource:

process 1	process 2

`seize(x);`
`x = x + 1;`
`release(x);`
	`seize(x);`
	`x = x + 1;`
	`release(x);`

- Intended meaning of `seize`:

  - once a seize of some resource has begun, no other process can begin to seize the same resource until it has been released.

Synchronization Narrows the Options

process 1	process 2
`seize(x);`	`seize(x);`
`x = x + 1;`	`x = x + 1;`
`release(x);`	`release(x);`

- Now only two possible traces:
  - `abcxyz` and `xyzabc`

A Bullet, but not a Silver One

- Synchronization takes different forms
- It can relieve dangers of shared access
  - Thereby promoting safety
- But it introduces new problems
  - New kinds of programming errors (e.g., forgetting to release a seized resource), and
  - Liveness problems

24.5 Liveness Properties

Liveness Properties

- Synchronization promotes *safety*: do we get a "correct" answer?
- But it may adversely affect *liveness*: the rate of progress toward an answer

. . . .

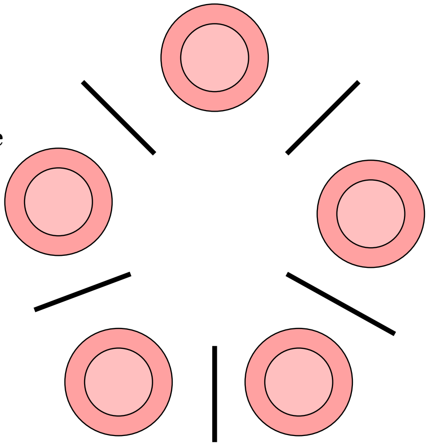
24.5.1 Deadlock

Deadlock

In *deadlock*, all processes are waiting on some shared resources, with none of them able to proceed.

. . . .

Example: The Dining Philosophers



- `N` philosophers sitting at a round table with `N` plates of spaghetti and `N` forks, one between each pair of philosophers.
  - Each philosopher alternately thinks and eats.
  - Each needs two forks to eat.
  - They put down their forks when thinking.

. . . .

Simulating the Philosphers

Represent each philosopher as an independent process:

```
loop
   pick up left fork;
   pick up right fork;
   eat;
   release forks;
   think
end loop;
```

- If all the philosophers are holding the left fork, the system deadlocks.
  - And the world has a few less philosophers in it!

. . . .

Demo

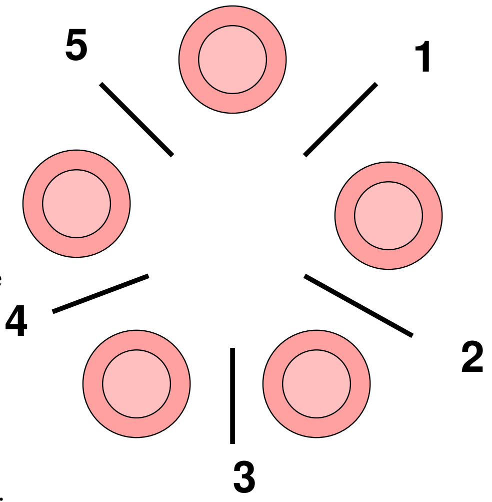
Try running Sun's Demo of the dining philosophers.

. . . .

You may need to play a bit with the timing control at the bottom, but after a while you will almost certainly see the system get into deadlock.

Avoiding Deadlock

- One way to avoid deadlock is a policy of "ordered resource allocation".
  - Number all the forks.
  - Require all the philosophers to pick up the lower-numbered of the two adjacent forks first.



. . . .

This is a "classic" fix from the world of operating systems, where this problem often arose in conjunction with different programs requesting access to I/O devices. If one program grabbed control of the printer and then requested a card reader, while another program had already gotten control of the card reader and was asking for the printer, deadlock would result. The classic solution was to assign each I/O device a unique number and require programs to request the devices in ascending order.

In general...

This works very nicely in this specialized case, but there is no general technique for avoiding deadlock in arbitrary situations.

- You usually have to rely on careful analysis by the program designers.

. . . .

24.5.2 Livelock

Livelock

A system is in *livelock* if at least one process is not waiting, but the system makes no progress.

```
loop
   pick up left fork;
   seize right fork
      if available;
   if seized then
     eat;
     release forks;
   else
     release left fork;
   end if;
   think;
 end loop;
```

This can get into livelock as each philosopher gets locked into a cycle of:

```
pick up left fork;
release left fork;
pick up left fork;
release left fork;
pick up left fork;
release left fork;
    ⋮
```

24.5.3 Fairness

Fairness

When multiple processes are waiting for a shared resource, and that resource becomes available, a *scheduler* must decide which of the waiting processes gets the resource.

- A scheduler is *unfair* if it consistently ignores certain processes.

What's Fair?

Precise definition of fairness is difficult. Some possibilities:

- Any process that wants to run will be able to do so within a finite amount of time. ("*finite-progress*")
- All processes awaiting a now-available resource have an equal chance of getting it.
  - Implies that it is possible for some processes to wait an arbitrarily long time.

Selfish Threads

Threads can contribute to unfairness by being "selfish":

```
class MyThread extends Thread {
  ⋮
  public void run()
  {
   while (true)
     ++counter;
  }
}
```

Unless the run-time system preempts this thread, it will hog the CPU.

. . . .

Yielding

```
class MyThread extends Thread {
  ⋮
  public void run()
  {
   while (true) {
     ++counter;
     Thread.yield();
   }
  }
}
```

Java allows threads to signal that they don't mind losing the CPU by `yield`ing to other threads.

. . . .

Thread Priority

Java also allows programs to set relative priorities for different threads:

```
Thread cpuHog = new Thread() { ... };
cpuHog.setPriority(Thread.MIN_PRIORITY);
cpuHog.start();
```

.... This could be useful if the GUI was sluggish because of the CPU cycles being burned up by this thread.

24.6 Safety

Safety

Concurrent programs are often non-deterministic.

- A *critical section* in a process is a section of code that must be treated as an atomic action if the program is to be correct.
- A concurrent program is safe if its processes are *mutually excluded* from being in two or more critical sections at the same time.
  - A fairly broad statement. Sometimes we can be more discerning by identifying groups of critical sections that can interfere with one another.
    - E.g., ones that work on a common shared resource

....

For example, if we have some critical sections that are writing data to the same output file, and other critical sections that are updating a shared display on the screen, our would be safe if processes were mutually excluded from the file output critical sections and mutually excluded from the screen update critical sections, but we might be able to tolerate one process writing to the file while another one updated the screen.

24.6.1 Mutual Exclusion Mechanisms

Semaphores

Semaphores

A *semaphore* is an ADT with two atomic operations, `seize` and `release`, and a hidden integer value.

....

seize()

- If the semaphore value is positive, it is decremented. The calling process continues.
- If the semaphore value is zero or negative, the calling process is blocked and must wait until the value becomes positive.

. . . .

release()

- The semaphore value is incremented. The calling process continues.
  - In practice, the scheduler often selects a process blocked on this semaphore to be allowed to immediately attempt to seize it.

. . . .

General Semaphores

- In its most general form, a semaphore can be initialized to any value `k`, thereby allowing up to `k` processes to seize it simultaneously.
  - Usually, we use *binary semaphores*, which are initialized to 1.
- Historically, the `seize` and `release` operations were originally called `p` and `v`.

. . . .

Yep, those were semaphores

process 1	process 2

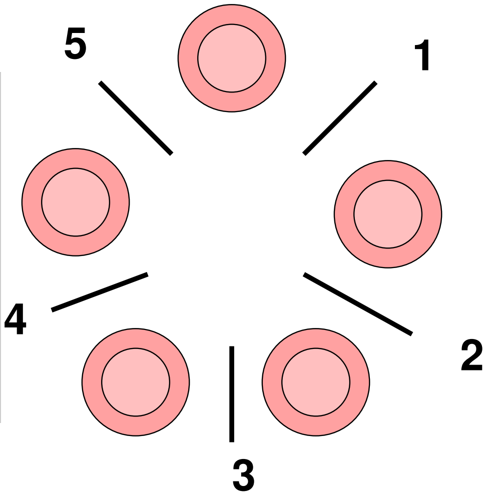
`seize(x);`
`x = x + 1;`
`release(x);`
	`seize(x);`
	`x = x + 1;`
	`release(x);`

- assuming that the semaphore is not “x” itself but is something associated with or labeled as “x”.

. . . .

Dining Philosophers with Semaphores

```
Semaphore fork[N];
Philosopher(i):
loop
   int first = min(i, i+1%N);
   int second = max(i, i+1%N);
   fork[first].seize();
   fork[second].seize();
   eat;
   fork[first].release();
   fork[second].release();
   think;
end loop;
```



. . . .

Recap

Semaphores are relatively low-level approach to synchronization

- Although available in Java, should not be used often

- Monitors and synchronized sections are more elegant

. . . .

24.6.2 Monitors

Monitors

A *monitor* is an ADT in which only one process at a time can execute any of its member functions.

- Thus all the ADT function bodies are treated as critical sections.

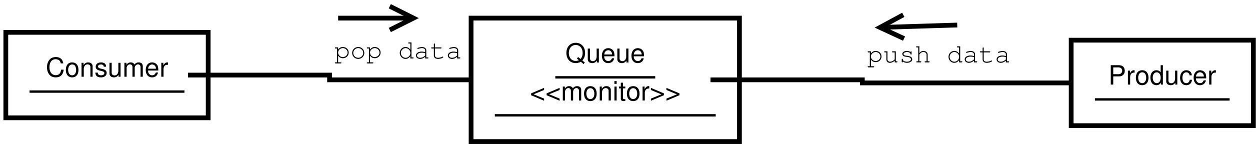
. . . .

A Common Pattern

- A program processes data from an input source that supplies data at a highly variable rate
  - Sometimes slower than the program processes data
  - Sometimes faster, often in bursts
- The program should not "freeze up" when data is unavailable
  - may have a GUI
- The data source must be checked often enough that inputs are not lost/ignored

. . . .

The Consumer-Producer Pattern



- Producer adds data to a queue
- Consumer removes data from a queue

- Each runs as a separate thread

What could go wrong?

. . . .

Synchronizing the Queue

We have seen earlier that queues are likely unsafe for simultaneous access.
We avoid these simultaneous update problems by making the queue synchronized.

. . . .

Monitored Queue

```
class MonitoredQueue {
   private ...
   public synchronized
      void enter(Object) {...}
   public synchronized
     Object front() {...}
   public synchronized
     void leave() {...}
   public synchronized
     boolean empty() {...}
   public
      int maxSize() {...}
 }
```

No two threads are allowed to be simultaneously in `synchronized` member functions of the *same* object.

. . . .

This is across the whole interface. e.g., One thread cannot be inside the `enter` function while another is in the `front()` function of that queue.

If we have multiple synchronized queue objects however, one thread could be adding to queue A while another thread is adding to queue B.

Java Monitors

Monitors are the preferred technique in Java, where they are created by marking functions as "synchronized".

- Each object of type `MonitoredQueue` is separately monitored.

```
    MonitoredQueue q1 = new MonitoredQueue();
    MonitoredQueue q2 = new MonitoredQueue();
```

- If process `P1` calls `q1.enter(x);`, then another process `P2` attempting to obtain `q1.front()` will be blocked until `P1`'s call to `enter` has completed.
- On the other hand, if process `P1` calls `q1.enter(x)`
  - Process `P2` can simultaneously execute `q1.maxSize();`

. . . .

Monitored Statement Blocks

The `synchronized` member function declaration is a special case of synchronized statement blocks

- Suppose we had only a regular queue and were not willing/able to change its interface.

  We could still achieve a safe solution by marking the critical statements in the code as `synchronized`:

```
class Consumer extends Thread {
   private Queue q;

  public void run() {
     while (true) {
         synchronized (q) {
           Customer c = q.front();
           q.leave();
         };
```

```
          ... process customer ...
        }
      }
   }
}

class Producer extends Thread {
   private Queue q;

  public void run() {
     while (true) {
       Customer c = fetchData();
       synchronized (line1) {
          q.enter(c);
       }
    }
}
```

. . . .

Synchronization on Objects

- When we mark statements as `synchronized`, we have to state explicitly what object we are synchronizing *on*.
  - When we mark member functions as `synchronized`, we don't need to do so because, implicitly, the object being synchronized on is `this`.
- *Important:* To actually get synchronization, blocks of statements must synchronize on the *same* object.

. . . .

24.7 Direct Control of Thread State

Waiting...

We often want to make a thread inactive until certain conditions are met.

This is the *bad* way to do it:

```
public void run() {
    while (true) {
        if (conditionIsMet()) {
            doSomethingUseful();
        }
    }
}
```

This is a *busy wait*

- We continuously burn CPU cycles testing the condition until it becomes true
  - Makes machine very sluggish

. . . .

Son of Busy Wait

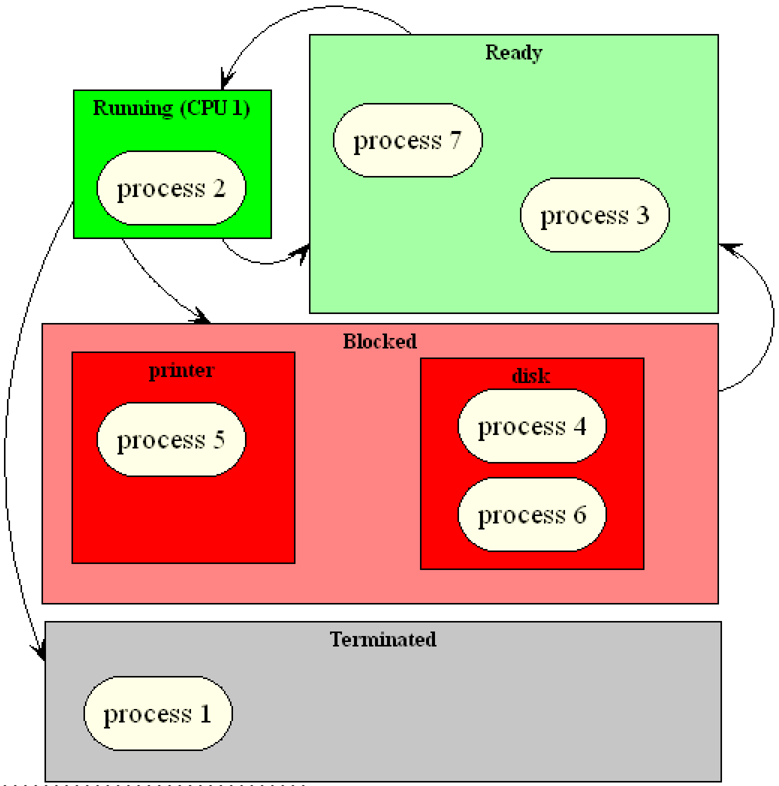
Only marginally better:

```
public void run() {
    while (true) {
        if (conditionIsMet()) {
            doSomethingUseful();
        } else {
          sleep(100); // wait 0.1 seconds
        }
    }
}
```

- still a busy wait
- still a bad idea

. . . .

Controlling the Process State



A better idea is to

- *block* the thread and then
- make it *ready* when the condition is met.
  - (has to be done by some other thread)

wait()

If we have a synchronized lock on some object x, then `x.wait()` will

- Put the thread on a "waiting for x queue"
- Make the thread ineligible for any more CPU time (*blocked*)

- Release the lock on x

. . . .

notifyAll()

If we have a synchronized lock on some object x, then `x.notifyAll()` will

- Make all threads on the "waiting for x queue" eligible for CPU time (*ready*)

. . . .

Example: Adding a Pause Button to ThreadArt

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JColorChooser;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;

/**
 * A simple example of GUI event handling in a Java application.
 *
 * This can be run as a main program or as an applet.
 *
 * @author zeil
 *
 */


public class ThreadArt2 extends JApplet {

    private boolean startedInAnApplet;

    // The Model
    private Color[] colors;
    private int stepSize = 5;

    private int cycleLength;
    private int cycleCounter;
    private boolean running;

    // The View & Controls
    private JFrame window;
    private JPanel canvas;
    private JButton colorChooser1;
    private JButton colorChooser2;
    private JButton pause;
    private JTextField stepSizeIn;

    private Animator colorChanger;


    private class ColorChooser implements ActionListener {
        private JButton button;
        private int colorNum;
        public ColorChooser (JButton button, int colorNum) {
            this.button = button;
            this.colorNum = colorNum;
        }

        @Override
        public void actionPerformed(ActionEvent arg0) {
            Color chosen = JColorChooser.showDialog(window, "Choose a color", colors[colorNum]);
            if (chosen != null) {
                colors[colorNum] = chosen;
                setColor (button, chosen);
                canvas.repaint();
            }
        }
    };


    /**
     * Thread that slowly changes the color of the drawing
     *
     */
    public class Animator extends Thread {

        public Animator()
        {
        }


        public void run()
        {
            running = true;
            while (true) {
                try {
                    sleep(50);
                } catch (InterruptedException e) {
                  break;
                }
                synchronized (this) {
                    while (!running) {
                        try {
                            wait();                                ❸
                        } catch (InterruptedException e) {
                            return;
                        }
                    }
                }
                cycleCounter = (cycleCounter + 1) % cycleLength;
                canvas.repaint();
            }
        }

    }




    public ThreadArt2()
    {
        startedInAnApplet = false;
        window = null;
        colors = new Color[2];
        colors[0] = Color.red;
        colors[1] = Color.blue;
        cycleLength = 100;
        cycleCounter = 0;
        running = true;
    }


    public static void main (String[] args)
    {
        ThreadArt2 instance = new ThreadArt2();
        instance.createAndShowGUI();
    }

    public void createAndShowGUI() {
        window = new JFrame();
        // set up the components
        window.getContentPane().setLayout (new BorderLayout());

        canvas = new JPanel () {
            public void paint (Graphics g) {
                super.paint(g);
                drawLines (g, getSize());
            }
        };
        canvas.setBackground(Color.white);
        window.getContentPane().add (canvas, BorderLayout.CENTER);
        canvas.setPreferredSize(new Dimension(400, 400));

        JPanel controls = new JPanel();
        colorChooser1 = new JButton("Color 1");
        controls.add (colorChooser1);
        setColor(colorChooser1, colors[0]);
        colorChooser1.addActionListener (new ColorChooser(colorChooser1, 0));

        colorChooser2 = new JButton("Color 2");
        controls.add (colorChooser2);
        setColor(colorChooser2, colors[1]);
        colorChooser2.addActionListener (new ColorChooser(colorChooser2, 1));

        stepSizeIn = new JTextField (""+stepSize, 5);
        controls.add (stepSizeIn);
        stepSizeIn.addActionListener (new ActionListener()
        {
            public void actionPerformed(ActionEvent e) {
                try {
                    Integer newSize = new Integer(stepSizeIn.getText());
                    stepSize = newSize.intValue();
                    canvas.repaint();
                } catch (Exception ex) {};
            }
        });

        pause = new JButton("Pause");              ❶
        controls.add (pause);
        pause.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {
                if (running) {
                    running = false;               ❷
                    pause.setText("Resume");
                    pause.repaint();
                } else {
                    synchronized (colorChanger) {
                        running = true;          ❹
                        pause.setText("Pause");
                        pause.repaint();
                        colorChanger.notifyAll();
                    }
                }
            }
        });


        window.getContentPane().add (controls, BorderLayout.SOUTH);

        window.setDefaultCloseOperation((startedInAnApplet) ? JFrame.DISPOSE_ON_CLOSE : JFrame.EXIT_ON_CLO


        colorChanger = new Animator();
        colorChanger.start();

        window.pack();
        window.setVisible(true);
    }

    /**
     * Sets the background color of a button to the indicated color.
     * Changes the foreground to wither black or white, depending on
     * which will give more contrast agasint the new background.
     *
     * @param button
     * @param color
     */
    private void setColor(JButton button, Color color) {
        button.setBackground(color);
        int brightness = color.getRed() + color.getGreen() + color.getBlue(); // max of 3*255
        if (brightness > 3*255/2) {
            // This is a fairly bright color. Use black lettering
            button.setForeground (Color.black);
        } else {
            // This is a fairly dark color. Use white lettering
            button.setForeground (Color.white);
        }
    }

    // Applet functions

    public void init() {
        startedInAnApplet = true;
    }

    public void start() {
        if (window == null)
            createAndShowGUI();
    }

    public void stop() {
    }

    public void destroy() {
    }
```

```
int interpolate (int x, int y, int i, int steps)
{
    return (i * x + (steps-i)*y) / steps;
}


Color interpolate(Color c1, Color c2, int i, int steps)
{
    return new Color (interpolate(c1.getRed(), c2.getRed(), i, steps),
            interpolate(c1.getGreen(), c2.getGreen(), i, steps),
            interpolate(c1.getBlue(), c2.getBlue(), i, steps));
}


class Point {
    double x;
    double y;
}

Point ptOnCircle (int degrees, int radius, Point center)
{
    Point p = new Point();
    double theta = Math.toRadians((double)degrees);
    p.x = center.x + (double)radius * Math.cos(theta);
    p.y = center.y + (double)radius * Math.sin(theta);
    return p;
}
```

```
public void drawLines(Graphics g, Dimension d)
{
    int dmin = (d.width < d.height) ? d.width : d.height;

    if (stepSize < 1)
        stepSize = 1;

    Point center = new Point();
    center.x = (double)d.width/2.0;
    center.y = (double)d.height/2.0;

    int k = Math.abs(cycleCounter - cycleLength/2);
    int theta = 60 * cycleCounter / cycleLength;

    for (int i = 0; i < 60; ++i) {
        int radius = dmin/2; //interpolate(dmin/4, dmin/2, k, cycleLength/2);
        Point origin = ptOnCircle(6*i+theta, radius, center);
        int j = i + stepSize;
        while (j >= 60)
            j -= 60;
        while (i != j) {
            Point destination = ptOnCircle(6*j+theta, radius, center);
            Color c = interpolate(colors[0], colors[1], k, cycleLength/2);
            g.setColor(c);
            g.drawLine ((int)origin.x, (int)origin.y,
                    (int)destination.x, (int)destination.y);
            j += stepSize;
            while (j >= 60)
                j -= 60;
        }
        }
    }



}
```

❶ A new button

❷ that sets *running* to `false`

❸ which causes the animation thread to block itself, putting it a queue associated with the object *colorChanger*

How does the thread ever get awakened?

❹ Next time we click the button, we set *running* to `true` and *notify* all threads (there should only be one) waiting on *colorChanger* to make themselves ready.

. . . .

Note the placement of wait() calls inside loops. That's a safety measure. If more than one thread is waiting to get something from the queue, and only one element is added to the queue, then all the waiting threads will be awakened by `notifyAll()`, but only one will actually get the data and the rest will immediately have to `wait()` again.

24.8 Summing Up

Making Things Parallel

- Divide the tasks into appropriate threads
- Examine the threads for shared variables (and other resources)
  - Consider whether simultaneous access to any of these shared resources could be unsafe.
- Add synchronization to prevent unsafe simultaneous access
- Analyze the synchronization for possible liveness problems

. . . .

Part VIII

Appendices

Appendix A

Syllabus

A.1 Basic Course Information

A.1.1 When and Where

Meets: Tues. and Thurs., 4:20-5:35, Dragas 1117

Website: http://www.cs.odu.edu/~zeil/cs330.html

A.1.2 Objectives:

This course will explore the techniques of object-oriented programming, analysis, and design. The emphasis will be upon the development of clean interfaces that permit easy modification and reuse of software components. Other techniques, drawn from outside the object-oriented approach, that significantly contribute to this goal will also be discussed.

Students will gain facility in an object-oriented programming language and will learn the constructs that differentiate such languages from others. This course will explore the idioms and styles of object-oriented programming in C++ and Java, with emphasis upon how these contribute to reusable software components.

Students will learn how to use object-oriented techniques in support of programming. In particular, students will be introduced to the process of object-oriented analysis as a means of understanding an unfamiliar problem domain. Students will learn to build and use models, expressed in the Unified Modeling Language (UML) to codify and evaluate that understanding and to evolve system requirements.. Students will learn how to use those models to facilitate a smooth transition from analysis to design and from there to implementation.

A.1.3 Textbooks:

Required:

- Horstmann, *Object-Oriented Design and Patterns*, 2nd ed., John Wiley & Sons, ISBN 0-471-74487-5
- Any reasonable C++ text that covers classes and inheritance (e.g., your CS250 or 333 textbook)

Optional:

- Fowler, *UML Distilled: A Brief Guide*, 3rd ed., Pearson AW, ISBN 0321193687

A.2 Communications

Steven J. Zeil	E&CS 3208
(757) 683-4928	Fax: (757) 683-4900
cs330@cs.odu.edu	

Specific course options and policies regarding communications are presented in Navigating The Course Website and Asking Questions.

A.2.1 Office Hours:

A week-by-week schedule of available meeting times with the instructor can be found at http://www.cs.odu.edu/~zeil/officehours/- . Note that the instructor is available during office hours both in person and, especially for distance students, via network conferencing and often has evening hours via network-conferencing only.

A.3 Course Prerequisites

The prerequisites for this course are:

- CS 250, Problem Solving and Programming II, or CS 333 Problem Solving and Programming in C++
- CS 252 Introduction to Unix for Programmers

or equivalents.

> If you need to review any of the prerequisite topics described below, the time to do so is early in the semester, *before* you need it to understand the lectures or are called to use it in assignments.

Students who have successfully completed those prerequisites should have acquired the following knowledge and skills:

A.3.1 General Programming Knowledge

Students should be familiar with certain basic programming techniques that are largely independent of any specific programming language:

- using editors, compilers and other basic software development tools.
- basic software design (i.e., stepwise refinement and top-down design)
- software testing
  - the use of scaffolding code (stubs and drivers)
  - selection of test cases for black-box testing
  - head to head testing
- debugging, including the use of debugging output, the use of automatic debuggers to set breakpoints and trace program execution, and the general process of reasoning backwards from failure locations to the faulty code responsible for the failure.

If you are uncertain about your preparation in any of these areas or if you simply wish to review some of them, you may want to visit the CS333 website. (For those of you entering this course from CS250, CS333 is an accelerated web-based course that combines material from both CS150 and 250.)

A.3.2 C++

I will assume that you are familiar with the basics of C++, including

- the various C++ statements and control-flow constructs,
- the built-in data types,
- the use of arrays, pointers, pointers to arrays, and linked lists,
- the use and writing of functions, and
- the basic use of structs and classes for implementing abstract data types.

Again, if you are uncertain about your preparation in any of these areas, you may want to visit the CS333 website.

A.3.3 Java

No prior knowledge of Java is expected. In general, CS students at the 300 level should be able to pick up new programming languages with only moderate effort. Because Java is closely related to C++, this is particularly true of students moving from C++ to Java.

If you believe that you need a more structured aid to learning Java, the 1-credit course CS382 presents the basics of the language in a self-paced form. In fact, much of the course material for CS382 will be listed as required reading for this course.

When the CS330 Outline page lists a section of 382 for readings, students are expected to both read the 382 lecture notes and to do the associated 382 labs. You will not be expected to do the 382 assignments or to take the 382 exam.

CS330 will pick up with Java topics more advanced than are currently covered in 382, including multi-threading and the development of GUIs and in general will explore how to use Java in a truly Object-Oriented style.

You may, if you wish, choose to register for CS382 and earn credit for it. In that case, you will be doing largely the same set of readings, but will then be obligated to do the assignments and exam for that course.

A.3.4 Unix

All students in the course will receive accounts on the CS Dept. Unix network, and knowledge of how to work with the Linux servers is part of the course prerequisites (CS 252). This course does not require familiarity with shell scripting. All other topics in CS 252 are required. Some assignments will require the use of software available only on the Linux servers. Others may require (or, at least, be simplified by) use of the X windowing system.

If you feel that you need review of this material, visit the CS252 website, linked above.

A.3.5 General Computer Literacy

You will be studying techniques in this course for preparing professional-quality software documentation. The key embedded word in "software documentation" is "document". Students taking this course should be able to use word processors and other common tools to produce good quality documents, including mixing text and graphics in a natural and professional manner.

A.4 Assignments

Assignments for this course will include *programming assignments* (in C++ and Java), which must be done on an individual basis, and *design assignments*, which may be done in small teams.

A.4.1 Computer Access:

Students will need an account on the CS Dept. Unix network to participate in this class. This account is unrelated to any University-wide account you may have from the ODU's Office of Computing and Communications Services (OCCS).

If you have had a CS Unix account in the recent past, you should find it still active with your login name, password, and files unchanged. If you have had an account and it has not been restored, contact the CS Dept systems staff at root@cs.odu.edu requesting that it be restored.

If you do not yet have such an account, go to the CS Dept. home page and look for "Account Creation" under "Online Services".

All students in this course are responsible for making sure they have a working CS Unix account prior to the first assignment.

Students will have access to the Dept's local network of Unix workstations and PC's in Dragas Hall and in the E&CS building. All students can access the Unix network and the Virtual PC Lab from off campus or from other computer labs on campus.

A.4.2 C++ compiler

The "official" compiler for this course is the Free Software Foundation's g++ (also known as gcc or GNU CC), version 4.5 or higher. This is the compiler that the instructor and/or grader will use in evaluating and grading projects. If you have access to other compilers, you may use them, but *you* are responsible for making sure that their projects can be compiled by the instructor and/or the course's grader using the official compiler.

You may want to develop your programs on the most convenient compiler and then port it over to the official environment. Please don't underestimate the amount of time that may be involved in coping with subtle differences among compilers.

You can do all work in this course using g++ on the CS Dept Unix servers via ssh/X. If you like, however, you can obtain the g++ compiler for free from a variety sources. Links will be provided on the course Library page.

A.4.3 Java compiler

The official compiler for this course is Java 1.6 from Oracle. For selected assignments, the instructor may impose limitations on the language and library features that can be employed (e.g., to maintain compatibility with the Java engines built in to the commonly available web browsers).

As with C++, this compiler will be available on the Dept Unix servers, but students may also download and install a free copy on their own machines via links provided on the Library page.

A.5 Course Policies

A.5.1 Due Dates and Late Submissions:

Late papers and projects and make-up exams will not normally be permitted.

Exceptions will be made only in situations of unusual and unforeseeable circumstances beyond the student's control, and such arrangements must be made prior to the due date in any situations where the conflict is foreseeable.

"I've fallen behind and can't catch up", "I'm having a busier semester than I expected", or "I registered for too many classes this semester" are *not* grounds for an extension.

Extensions to due dates will not be granted simply to allow "porting" from one system to another. "But I had it working on my office PC!" is not an acceptable excuse.

A.5.2 Academic Honesty:

Everything turned in for grading in this course must be your own work. Some assignments may be done by small teams, in which case the submitted material must be the work of only those team members. Such assignments will be clearly marked as team assignments. Where teams are permitted, specific guidelines will be given in the online assignment description. In the absence of any such *explicit* statement, an assignment must be performed by a single individual.

The instructor reserves the right to question a student orally or in writing and to use his evaluation of the student's understanding of the assignment and of the submitted solution as evidence of cheating. Violations will be reported to Student Judicial Affairs for consideration for punitive action.

Students who contribute to violations by sharing their code/designs with others may be subject to the same penalties. *Students are expected to use standard Unix protection mechanisms (`chmod`) to keep their assignments from being read by their classmates. Failure to do so will result in grade penalties, at the very least.*

This policy is *not* intended to prevent students from providing legitimate assistance to one another. Students are encouraged to seek/provide one another aid in learning to use the operating system, in issues pertaining to the programming language, or to general issues relating to the course subject matter. Student discussions should avoid, however, explicit discussion of approaches to solving a particular programming assignment, and under no circumstances should students show one another their code for an ongoing assignment, nor discuss such code in detail.

A.5.3 Attendance:

Attendance at classes is not generally required, but students are responsible for all material covered and announcements made in class. Consequently, if you are going to miss class, be sure to get notes, handouts, etc., from another class member.

A.5.4 Grading:

Assignments:	40%
Midterm Exam:	25%
Final Exam:	35%

It is my general policy that, should a student perform significantly better [1] on the final than upon the midterm, or should a student have one project grade that is significantly lower than the rest, to waive that single low grade (adjusting the percentages

[1] a rise in class rank of at least 1.25 standard deviations

of the remaining grades accordingly).

For the truly curious, some further information on my approach to grading is available.

A.5.5 Topics

Pre-OO: ADTs and Classes A review of the concept of Abstract Data Types and the use of C++ classes to implement them.

OOAD (Object-Oriented Analysis and Design) 1: Domain Models The Unified Process Model, reading common documents to discover classes, modeling an application domain with CRC cards and UML class relationship diagrams

OOP (Object-Oriented Programming) 1: C++ Inheritance and dynamic binding in C++

OOAD 2: Analysis Models: Use-case scenarios, modeling with UML interaction diagrams

OOP 2: Java Introduction to Java, self-checking tests, inheritance and subtyping in Java

OOP 3: Applying OO Techniques Container classes, Graphic User Interfaces, multi-threading

OOAD 3: UML Models: Making the transition from analysis to design to implementation

Appendix B

Navigating the Course Website

B.1 Overview

This course website is quite large. By the end of the semester, there will be thousands of pages of material. It can be a bit daunting when you are starting out. This document explains the basic organization of the site.

It's worth pointing out that, this semester, I am trying out a number of new things with the site organization. I'd appreciate feedback in the Forum regarding what is and what is not working well for you.

The course website is split over two different web servers.

- The ODU Blackboard system is used for announcements, the course Forum, exams, and calendar. All of these things are presumed to be of interest only to students actually registered for the course this semester. You will need to log in using your ODU Midas account info to access this material.
- The CS Dept.'s secure web server is used for slides & lecture notes, and policy documents like this one - this is material that I leave open for anyonw to read, and requires no login.

  It will also house assignments & solutions to assignments. You will need to log in with your CS Dept. network account info to access this material.

Grade reports will probably be hosted on the CS Dept server as well – I have not worked out the details of this yet.

B.2 On Blackboard

When you enter the course from Blackboard, you will start at the Announcements page.

From there, your main options are:

Syllabus & Policies, Topics, Library These take you to the respective pages on the CS server, which are described in more detail later.

Forum The course Forum is for open discussion and commentary related to the course. The course communications policies are described in a later section.

At the start of the semester, there will be only a single Forum for general discussion of course matters, open to all students. Later in the semester, when we start on team projects, I may create separate Forums for each team, readable only by the team members.

Calendar Access to a calendar listing significant upcoming events.

Exams Access to course exams – this will not be visible until shortly before the midterm exam is posted.

B.3 On the CS Server

You'll probably spend most of your time with the various documents on this server.

The documents here are a mixture of HTML pages and PDF documents. The HTML is designed to be basic enough that it should be viewable on almost any browser.

B.3.1 Common Elements

At the bottom of almost every page (HTML or PDF) you will see the following symbols:

- 🏠 Clicking this takes you to the home (Topics) page for the course.

✉ Clicking this opens up a new email message to me, pre-loaded with the title and URL of the document on which you clicked in the message body and the name of the course in the subject line. (See figure B.1 for an example.)

I **strongly** encourage you to use this method for asking questions that aren't going into the Forum. In particular, questions about difficulties you are having solving an assignment should never be put into the Forum (because it is open to all students) but should be asked via email.

Using this symbol/link guarantees that you are going to be sending to the correct email address, and that I will know what course you are asking about and what document you were looking at when you came up with your question.

Please take the time to ask from the most appropriate page. If you have a question about an assignment, for example, go to the assignment page and use the ✉there. Don't ask your question from the Topics page. You won't receive a meaningful response as quickly if my first reply to you is simply asking what you are talking about.

Although the HTML pages are simple enough to be handled properly by almost any browsers, these common symbols on the PDF pages are a little trickier. A few PDF viewers don't handle links to web URLs at all. Some will handle simple links (e.g., the ⌂ symbol) but won't handle `mailto:` links (the ✉ symbol) at all or will ignore the portion of the link that pre-loads the subject line and message body.

You might want to try those symbols out, at the bottom of this page, right now to check and see what your PDF viewer does.

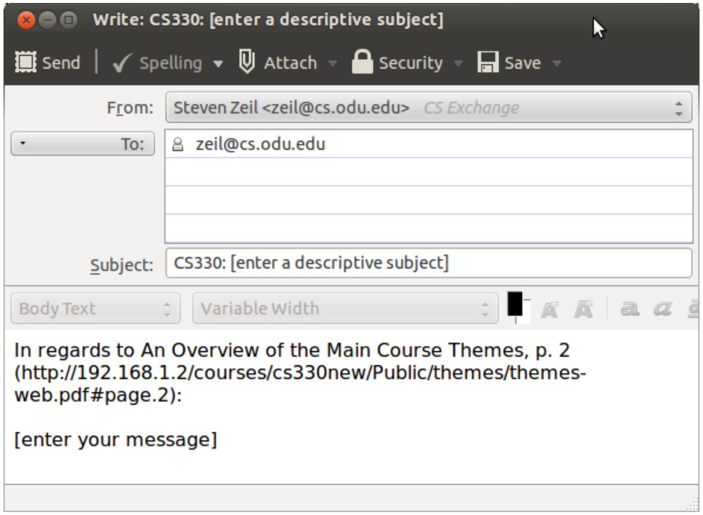
B.3.2 The Directory Pages

The documents on the CS server are organized via a collection of several *directory pages*, easily recognized by the row of buttons down the left side. The *directory pages*, which can be recognized by the row of buttons down the left side, serve as gateways to main content of the web site.

- The Topics page presents the outline of the course subject matter together with textbook readings, assignments, lecture notes or slides, etc.

  Each link is accompanied by an icon identifying what kind of document it takes you to: lectures, labs, assignments, etc. There is a key at the the bottom of the page explaining the icons.

- The Policies page presents the course syllabus and other documents relating to course policies and procedures and on how to get started working in the course. At the start of the semester, you should make a point of getting familiar with everything on this page.



From: Steven Zeil <zeil@cs.odu.edu> CS Exchange

To: zeil@cs.odu.edu

Subject: CS330: [enter a descriptive subject]

In regards to An Overview of the Main Course Themes, p. 2 (http://192.168.1.2/courses/cs330new/Public/themes/themes-web.pdf#page.2):

[enter your message]

Figure B.1: Email pop-up with some content pre-loaded

- The Library page contains links to a variety of reference materials and software packages that may prove useful to you in this course.
- The Blackboard button takes you to the Announcements page on Blackboard.
- The Forum button takes you to the main course Forum on Blackboard.
- The Book button (available later in the semester) will take you to a complete collection of all lecture notes in one document.

B.3.3 PDF Documents

Formats

Most of the course content is in PDF. Most of those PDF documents are course lecture notes and slides. PDF offers advantages over HTML of allowing me precise control over the appearance of math, program listings, and other content important to CS courses, and allowing me to know that it will appear the same no9 matter what system it is viewed on. PDF documents are also more suited to off-line viewing, which may be useful for people with tablets/laptops wandering in and out of wireless coverage areas.

PDF is, however, a page-based format. And, for tablets and other small-screen devices, PDF works best if the page size is matched to the screen size. So, in something of an experiment this semester, when you select a link (e.g., like this one) to one of those documents, you will be taken first to a web page listing the available formats:

slides Mainly for me, in class

web A format chosen for viewing on desktop and larger laptop screens, where you probably don't want to fill the whole screen but want something that can be read without excessive eye & neck movement.

printable Formatted for 8.5 x 11 inch paper, for those who simply *must* have printed copies.[1]

[1] If you are using printed copies just to scribble notes on, you might be interested to know that there are some very good free PDF annotation programs that allow you to draw on, type in the margins of, or add "sticky notes" to a PDF file that you have downloaded.

tablet formats Formats for 7 and 10 inch screen devices, with aspect ratios of both 4:3 and 16:10.

I'm interested in getting peoples' feedback on these in the Forum. Do you find them useful or not? Are the fonts too small or too large? Do you actually use them off-line, or is it OK to have links to on-line content (e.g., longer code listings that might be harder to read when split across multiple pages).

Content

One thing that you may notice fairly quickly is that, for the lecture notes, most of the available formats have more content than is present in just the slides.

As you may be aware, I offer this course as both a "live", traditional lecture course and as a webcourse in alternating semesters. The webcourse was created initially by my going through my slides from the live course and adding text that approximated what I would have said when presenting each slide. As semesters have gone by, when I wanted to change something in the lectures, sometimes I would change the liver version's slides first, sometimes I would change the web version's text first. Then I would have to remember (usually during the following semester) to make the corresponding changes to the other version of the course. This was getting to be a pain.

So, now, I have a single source document containing both the slides and my commentary. The "slides" format contains only the slides. The other formats contain both.

Linking

Within the PDF documents, you will find several kinds of links.

- This is a link to another page within this same document.
- This to an web page outside the course.
- This is a link to another document (the syllabus) within the course.

  Links like this will preserve your chosen document format. If you are currently, for example, viewing a format for 10 inch, 4:3 ratio screen, likes like the one above will take you to the 10 inch, 4:3 ratio version of the indicated document.
- Sometimes you will see a link that looks like this. This is also a link to another document within the course. The ✍ symbol is a warning that the link is to some specific place inside the document, rather than to the document as a whole.

Not all PDF viewers will open to the indicated page. Some will insist upon opening at the first page.[2] Or, if you had already opened that document recently, some will always take you to the last page you looked at in that document.

So links like this are accompanied by a page number as part of the visible link. If your browser/PDF viewer don't jump to the desired location, you at least know where you should be paging to.

B.4 Communications

Other than meeting with me during office hours, your main options for communicating with me are e-mail and the Forum.

B.4.1 Forum vs. E-Mail

Questions about how to solve a graded assignment or questions about solutions to exam questions should be sent by e-mail. They should not be put into the Forum.

Questions about course policies, about the subject matter of the lectures and readings, or about wording/interpretation of the assignments may be asked via e-mail or posted in the Forum.

If you aren't sure whether a question qualifies for the Forum or not, ask it via e-mail. If I receive an e-mailed question that I think may be of general interest, I may copy it to the Forum and answer it there.

B.4.2 Asking Good Questions

A question is the beginning of a dialog. A well-prepared question will get you an informative answer quickly. A poorly-prepared one may get you irrelevant answers or may require several rounds of back-and-forth dialog, delaying your eventual answer by many hours or even days. So it's in *your own self-interest* to ask your question in a way that gets you the answer you need as quickly as possible.

Identification

[2] Sometimes it's not the PDF viewer that is the problem, but the web browser plug-in used to launch the PDF viewer.

Who are you? If you are sending me email, make sure your course login name or your real name appears somewhere in the message. I hate getting mail from `partyAnimal@hotmail.com` saying "Why did I get such a low grade on question 5?" when I have no idea who this person is!

What course is this? Again, if you are sending me a question via email, please remember to state which course you are asking about. I teach multiple courses most semesters, and having to go look up your name to see which of my courses you are talking about is annoying. In fact, it's a good idea to make the course number part of the subject line.

It helps if you are using the ✉ links at the bottom of the course pages, because that automatically puts the course name into the subject line.

Which brings us to the next item…

Use a clear and precise subject header. In e-mail, A good subject header helps me find your message later if we need to do 2 or more rounds of back-and-forth responses. It's quite annoying when someone writes "*As I mentioned in my last message, …*", and I then discover that the subject header is empty or contains only the course number, making it one of hundreds with the same subject tag! Empty or ambiguous email headers also increase your chances of being flagged by my spam filters.

In the Forum, a clear subject line not only helps me know what you are talking about, but it is a basic courtesy to your classmates, letting them know whether they should bother reading the thread.

A good subject line distinguishes your topic from other messages that your classmates might send. "Question" or "Assignment 3" are *not* good subject lines.

What are You Talking About? If your question is related to some document (a set of lecture notes, an assignment), tell me what document and, for multi-page document, what part of the document you are asking about.

Again, for e-mail it's easiest if you use the ✉ links because they fill that information in to your message. For a Forum post, you will have to supply that information yourself.[3]

How do you identify an internal document location? One way is by document URL plus page number. Alternatively, you can give the document title and describe the location (section & subsection numbers, quote a bit of distinctive text, etc.).

Be aware that you *can't* get by with the combination of document title and page number. That's because I'm providing these documents in multiple formats, and the same text is likely to be on very different pages in each format.

[3]You might find it convenient to use the ✉ link and copy-and-pasting the document info from the message body into your post.

Thou Shalt Not Paraphrase

There's nothing more frustrating than getting a question like

> *"When I try to compile my solution to the first assignment, I get an error message. What's wrong?"*

Grrr. What was the (exact) text of the error message? Was this on a Linux or Windows machine? What compiler were you using? What compiler options did you set? What did the code look like that was flagged by the message?

No, I'm not kidding. I get messages like this all the time. And it wastes my time as a question answerer to have to prompt for all the necessary information. It also means a significant delay to the student in getting an answer, because we have to go through multiple exchanges of messages before I even understand the question.

The single most important thing you can do to speed answers to your questions is to be specific. I'm not psychic. I can only respond to the information you provide to me.

- *Never, ever paraphrase an error message* ("*I got a message that said something about a bad type.*"),
- *Never, ever paraphrase a command that you typed in that gave unexpected results* ("*I tried lots of different compilation options but none of them worked.*")
- *Never, ever paraphrase your source code* ("`I tried adding a loop, but it didn't help.`")
- *Never, ever paraphrase your test data* ("*My program works perfectly when I run it.*")

All of the above are real quotes. And they are not at all rare.

The problem with all of these is that they omit the details that would let me diagnose the problem.

And it's not all that hard to provide that info. Error messages can be copied and pasted into your e-mail or Forum posting. The commands you typed and the responses you received can be copied-and-pasted from your ssh/xterm session into your email or posting. Your source code can be copied-and-pasted or attached to e-mail and posts. [4]

[4] Note that this information is almost always plain text. Unless you *really* need to show me graphics, please avoid screen shots. They are often hard to read and often do not allow me to make the fine distinctions I need to tell what is going on. Keep in mind that raster graphics formats (gif, jpg, png, etc.) often look very different when rendered on screens with different resolutions.

If I Ask You a Question, Answer It

I often respond to a student's question with further questions of my own[5], sometimes to get more info I need, sometimes to guide the student towards an answer I think they should be able to find for themselves.

It's surprising how often students ignore my questions and either never respond at all, respond as if my questions were rhetorical, or, if I have asked 2 or 3 questions, pick the one that's easiest to answer and ignore the rest.

This pretty much guarantees that the dialog will grind to a halt as I wind up repeating myself, asking the same questions as before, and some students go right on ignoring my questions, ...

B.4.3 Netiquette

A few basic rules for both e-mail and the Forum:

Be civil: Communications via e-mail and the Forum are expected to conform to the norms for civility and respect for ones' classmates and instructors that are common to all on-campus speech and writing.

Be tolerant: Short typed messages are not very good ways to convey emotional content. Many people tend to read insults and slights where none are intended.

A short, terse response probably does not mean that someone is disrespecting you or is unwilling to give your post due consideration. It's more likely to mean that someone was trying hard to quickly get some sort of useful response back to you rather than forcing you to wait until they had time for a lengthier one.

I often respond with links to relevant readings. Please don't get all huffy because you've already read it. Unless you *told* me in your earlier posts that you had read it, I can't assume that. Students often miss things in the readings. And sometimes pointing out that a specific page **is** relevant can make all the difference in people's understanding.

Stay relevant: If you see an existing discussion thread about an assignment or lesson for which you have a very different question, don't just jump in with a reply in that thread. *Hijacking* someone else's thread is considered rude, and it leads to confusion because it can become unclear whether subsequent replies are addressing your new issue or the one that the thread was originally discussing. Start your own thread instead.

[5] Teachers since Socrates have always done this, and students have always been annoyed at it. But who are we to argue with history?

Appendix C

Preparing and Submitting Assignments

This document explains the procedures for turning in assignments in this course.

C.1 Programming Assignments

Why are the Programs used in Assignments so Big? In one sense, Object-Oriented programming is all about organizing the abstract data types that make up a program. A program that's small enough to need only a few abstract data types doesn't need OO, won't benefit from it, and can't be used to practice OO. That creates a bit of a dilemma for an instructor when it comes to composing assignments. If you're really supposed to get a feel for how OO works, it can't be on programs that are small enough for you to write in a week or so.

Most of the assignments in this course will therefore involve making small changes to a set of code for a nearly complete program, or altering a completely working program to take advantage of some new data structure or algorithm we have studied. As it happens, that's probably a more realistic approach. Most real-world programming projects will not involve creating an entire program from scratch. Far more often, you will be asked to modify, fix, or enhance existing code.

You yourself will probably not be writing that much code in any given assignment, But part of the challenge in working with programs at this scale is that you have to develop a knack for reading code and seeing where you need to make changes and what you need to leave alone.

Grading is performed on the CS Dept's Linux servers. You may do your development work on the CS Dept's PCs, or on other machines more convenient to you. It's a good idea, though, before you submit your work, to upload your code to the CS Unix network and check to be sure that it still works after doing so. This is not something you should take for granted!

The Grading Criteria for Programming Assignments

1. Satisfy the stipulations and limitations of the assignment.

   Each assignment will have certain stipulations. You may be limited in what files you can change, what data structures you can use, etc. The point of any assignment is to practice and demonstrate your ability with certain skills and techniques that we have covered in class.

   You might be able to get that program running in some completely different manner, maybe rewriting the whole thing from scratch. It might even be easier for you to do it that way. That doesn't matter. If you have not employed the stipulated techniques, your submission will be disqualified.

2. Produce the correct output.

   The primary scoring criteria for programming projects is the percentage of the test cases on which your code produces correct output.

   While it's common in beginning programming classes to write interactive code that prompts for inputs and accepts them from the keyboard, few programs these days work that way. Most truly interactive programs will work through a graphic user interface. And we will do those late in the semester. But there *are* still many programs that work through simple text input/output interfaces. These programs typically read and write to files or possibly directly to other programs. Consequently, *the I/O formats of these programs are crucial and must be followed very precisely*.

The Anti-Grading Criteria for Programming Assignments These are things that you will *not* get points for in the grading of programming assignments:

1. The code is "almost correct" or "looks similar to" a correct solution.

   Sorry, but almost everyone who submits buggy code thinks that their code is "almost correct", that it "looks like" code that "should work". It's a well-known truism in our field that most software projects are 80% complete for 90% of their development time. It's very easy to get code to a point where it looks like something that should be correct and yet fails every test case submitted to it. The real effort in programming, and what you are rewarded for in the grading, is for getting past that point to something that actually passes one or more tests.

This does not mean, as some folks complain, that I don't give partial credit on programming assignments. Some credit is awarded for code that compiles. More importantly, I generally have multiple test cases, and you will get credit for those that you pass, even if you fail some others.

2. Time spent on the assignment

   "I spent so many hours on this assignment." That's gratifying, but do you think that your classmates did not? Or, if someone managed to produce correct output while spending only half the time that you did, do you think that I should take points away from them for having done it more efficiently than you?

   Telling me how much time you spent on an assignment may earn you sympathy. And in some cases I may want to talk with you about whether you are using your time in a smart way. I may be able to suggest more efficient ways for you to approach your coding, testing, and debugging. But it does not demonstrate your mastery of whatever techniques the assignment is trying to showcase.

The Auto-Grader Most programs in this course will be graded automatically. You will submit your assignments electronically. Within a limited amount of time (typically 15-30 minutes, but this may vary depending upon how many other people are submitting) your code will be compiled. If it compiles successfully, it will be run on the instructor's test data and scored on how well it performs. The results of the compilation and testing will be emailed to you (at your CS email account) and will be available on the website.

The point of the auto-grader is to provide you quick feedback. In many cases, you may be able to use that feedback to fix things before the final due date of the assignment. If I graded in the "traditional" manner (you turn things in and I grade them after the due date), you would not get that opportunity. That can be very frustrating (to me as well as to the students) when students make "silly" mistakes such as submitting the wrong file, writing code that takes its input from the wrong source, writing output in an incorrect format, etc.

Some people complain that the auto-grader is too picky or too strict. That's a bit misguided. The auto-grader only runs the tests that I would have run if I were executing your program "manually". (Now, if you want to complain that *I* am too picky, that's another matter entirely.)

Multiple Submissions *If there is time before the due date, programs may be resubmitted without penalty at most twice (after the first submission, for a maximum of three total submissions).* A few programming assignments may not permit resubmissions, in which case that will be stated explicitly as part of the assignment description. Non-programming assignments, in general, cannot be resubmitted.

The reason for allowing resubmissions is to give you a chance to recover from "silly mistakes" as described earlier or from fundamental misunderstandings about what the program is supposed to be doing. It is not intended to allow you to keep chasing points by passing one more test each time. You are expected to test your own code before submitting, and, if you have done that, you should not actually be surprised by the percentage of my tests that you pass or fail. Now, if you did not make a silly mistake and have time left before the due date and can incrementally improve your percentage of tests passed, I'm not going to complain if you use the allowed resubmissions for that purpose. But I'm also not going to be sympathetic to requests for "just one more submission" in order for you to increase your percentage of tests passed.

Appeals Instructors are fallible too, and sometimes we will have a bad test. If you discover a bad test when you review the assignment solution, you can ask to have it re-scored.

If you have done multiple submissions and actually made your code score lower on your final submission than it had on an earlier one, you may ask that your score be rolled back to the higher score. You have to ask, though. I generally don't notice when this has happened unless you tell me.

On the other hand, i

C.1.1 Checking Your Code

Compiling

The official C++ compiler for this course is the Free Software Foundation's GNU g++ compiler, version 4.5. This compiler is available on the CS Dept's Linux workstations and Vista PCs (as the Code::Blocks package). You can also download this compiler for free and install it on your own PC.

Do *not* use Dev-C++, even if you used it in a prior class. Its version of the GNU g++ compiler is outdated.

Do *not* use the Microsoft Visual compiler. It has numerous bugs that make it incompatible with much of the code that will be used in this course.

It's important that you not only use the same compiler that we will use to grade your code, but that you compile your code the same way we will be compiling it - using the same commands and options. This will be easy for you to check. Each assignment will be supplied with a `makefile`. You can compile your code on a Linux system via the command

```
make
```

.

Executing

In most cases, the assignment description will provide details on how the program should be executed (i.e., what command line parameters to supply, where to find inputs and outputs). You may also find details on this provided in comments in the .cpp file containing the `main` function.

Most assignments will describe a sample input and its expected output. These may be provided as sample files in the assignment directory.

Testing

You are responsible for testing your code before submitting it. If you do a poor job of this, you should not be surprised if my tests find bugs that you were unaware of.

- With that said, my tests are seldom particularly subtle or difficult. Most of them follow the basic forms of black-box testing with which you should already be familiar. In some cases, you may need to write your own test drivers or other scaffolding.
- How can you tell if your output is correct? In most assignments, I will supply you with a compiled version of my own solution. You can run your program and mine on the same input and compare the outputs in head-to-head testing. Consequently, there should be very little question as to what output is expected for any given test input.
  - You will find these compiled versions of my solution in the bin directory of the assignment. Within that directory, you will find directories `Linux` and `Windows`, each containing an executable program.
    - Be sure that you choose the version appropriate to the machine you are running it on. You cannot execute a program compiled for Windows on a Linux machine or vice-versa!
    - The Linux versions are compiled for our CS Linux (Ubuntu) servers, available at `linux.cs.odu.edu`. They cannot be run under most other Unix variants. *Stay away from the Solaris ("fast") machines.* They'll just cause you problems in this course.
  - Note also that you do not need to copy the Linux program to your own directory in order to run it. (Doing so may actually cause confusion by overwriting the compiled executable from your own code.) Just run it via its full path:

    ```
    ~zeil/cs361/Assignments/assignment/bin/Linux/executableName any-arguments-required
    ```

    And remember, when typing long path names in Linux shells, the Tab key can be used to auto-complete a file or directory name after you give the first few letters.

- Finally, do not trust that because your code works on a PC that it will work on Linux as well. Among other potential problems, my own experience is that Windows is more likely to provide newly allocated memory filled with zeros. This tends to hide errors involving uninitialized variables, missing return statements, and some pointer manipulation errors.

Submitting

Each assignment page will have a button or a link that you can use to submit your solution. You can upload files either from file system the machine at which you are browsing or directly from your CS Unix account.

The submissions page will also allow you to check to see if a prior submission has been graded and to view the most recent grade report. In most cases, you should have received this report by e-mail, but this provides you a backup in case the email message disappears into the Internet ether.

Afterwards

When you have completed an assignment, you can return to the submissions page and ask to see the solution. You will see not only my solution to the assignment but also the tests that I used for the evaluation. These tests are sometimes simple input files. Other times they are shell scripts that can be executed to run the test.

Even if (maybe especially if) you do poorly on an assignment, you can still learn from it by reviewing the solution. I assume that everyone does so. I'm happy to answer questions about assignment solutions, but if you do not ask, I will assume that you read the solution and understood it.

C.1.2 Files Provided in Assignment Directories

Each programming assignment is found in a separate directory, specified on the assignment web page. You will need to copy the files from that directory into your own working directory.

Make sure your files are protected! It is your responsibility to protect your assignments and other course work from access by your fellow students. Of course, they shouldn't be looking there, but the ODU honor code recognizes that making your work freely available for copying is also unethical behavior.

Leaving your assignment solution files unprotected for your classmates to read is considered a violation of the ODU honor code and may result in sanctions up to and including expulsion from the University.

Typically, each assignment directory will include a number of .cpp and .h files. Some of these are for you to modify, some should be left alone. You can tell which should be modified by consulting the list of files to be submitted. Anything that you aren't going to turn in *must* be left unchanged.

Other files that you may find in the assignment directories include:

makefile This is a standard Unix tool for project management. If an assignment comes with a `makefile`, then you can compile the program by giving the command `make` and you can clean up your directory when the project is done by giving the command `make clean`

make.dep, *.d These files support the `makefile`. You can ignore them. If you get a warning message about a "missing make.dep", ignore it. The `make.dep` file will be created automatically.

executable program In many cases, I will provide you with a compiled version of my solution. You can use this as an aid to testing your program by running your version and mine on the same inputs and comparing the outputs to see if they agree.

Important: Executable files can only be run on the same CPU model and operating system for which they were compiled. I typically provide, in the `bin` directory, one executable for Windows PCs and one for our Linux servers.

***assignmentName*.jar** This is the compiled version of my solution for Java programs. Typically, the program can be run via the command

```
java -jar assignmentName.jar parameters
```

where the *parameters* are any command line parameters required by the program itself. If the program requires no command line parameters, the the program can probably be executed on a Windows or other GUI-based system by double-clicking the `.jar` file in a file explorer display.

***assignmentName*.zip** Because Java often requires source code files to be arranged in subdirectories of your project, some assignments may provide source code files already arranged in the appropriate file structure. In that case, the source will be provided as a `.zip` file, that you can copy and then unpack into your own working directory.In many cases,

C.1.3 Submission guidelines

- Pay attention to the list of files you are expected to submit. Under no circumstances should you ever turn in compiled object code (.o or .obj) or executables (.exe).

  The one thing that you may turn in, in addition to the files explicitly listed in the assignment's submission list, is an optional file named README.txt, in which you may include any special notes to the grader that you feel may be necessary.

- Be very sure you have the correct names on all files. In particular, note that Unix file names are case-sensitive: "foo.cpp" and "Foo.cpp" are not equivalent, as they are on some Windows file systems. If the names are wrong, your program will not compile. If it doesn't compile on the Unix system, it's wrong. "But I had it working on my home PC!" is not an acceptable excuse.

- Do not change any other files provided with the assignment (except for your own temporary testing/debugging purposes). In particular, beware of changing interfaces! The instructor's test drivers may or may not be identical to the code provided with the assignment, but it will certainly assume that you have not altered the function names, parameter lists, or other visible interfaces. In the end, the files that you submit must compile with the *original* versions of the other files provided with the assignment.

- Pay close attention to the expected format of the output. Output that is printed in a different order, in an incorrect format, has extra or missing characters, or in which you've inserted extra debugging information, is *wrong*.

  If this seems a bit harsh, consider that, in the real world, programs must talk to each other, and to do that they must agree on I/O formats. Using an incorrect output format in those circumstances will not only break the communications path between the programs, but will earn you the enmity of your fellow programmers because management may suspect that it was *their* input code that failed rather than your output code.

C.1.4 After the assignment is over

Assignments for this course are supposed to be a learning experience. If you are able to complete the assignment successfully, then you have presumably learned what I intended from the experience of working out your solution.

But if you were unsuccessful or only partially successful at the assignment, it's still important that you try to learn from it. The first step in doing so is to return to the assignment submission page and click on the "View Solution" button. You will see not only my solution to the assignment but also the tests that I used for the evaluation. These tests are sometimes simple input files. Other times they are shell scripts that can be executed to run the test.

Study my solution and compare it to your own. Try some of the test data listed there.

If, after doing that, you feel that you understand what was involved, then you are ready to move on. If not, it's time to ask questions.

I'm more than happy to take questions, but if I don't hear from you, I will assume that you viewed the solution and understood it.

C.2 Non-Programming Assignments

When asked to write essays or reports or to answer questions, you should prepare a document with your answer, using your favorite text editor or word processor, and then use the "Submit" button on the assignment page to submit that file(s). Normally, each assignment will call for a single document.

Note that, in this course, the non-programming assignments generally revolve around preparing documentation of models, designs, etc. Documentation is all about *communication*. Anything that gets in the way of communication, including careless organization, missing information, unreadable graphics, or non-portable document formats, is therefore likely to be penalized.

Unless otherwise instructed, your document must be in one of the following formats:

- plain ASCII text, saved as a .txt file.[1]

- Adobe Portable Document Format (PDF), saved as a .pdf file.

  You can produce these if your system has the Adobe Distiller software installed. Some word processors (notable the free OpenOffice suite) allow you to generate PDF directly.

  You can also produce PDF from any Windows program if you have installed PDF Creator or a similar program that adds a "pdf file printer" to your list of available printers. See the course Library page for more details.

Do not assume that I will accept other document formats unless you have checked with me first! You will lose 10% of the assignment value for submitting in an unapproved format, 20% if you force me to switch to a different machine that has software capable of reading your document, and 100% if I cannot find such a machine.

[1] In some instances, this will not be acceptable. For example, if you are told to prepare a report that is to include graphs or other images, you obviously cannot submit in plain ASCII text.

C.2.1 Graphics and Drawings

In assignments that ask you to prepare charts or diagrams, you should use the appropriate drawing tools (you will find a list of some recommended ones on the Library page) to prepare your graphics.

These should then be copied and pasted into a word processing document, submitted in one of the approved formats listed earlier. Do *not* try to submit multiple separate graphics files.

The document should clearly indicate problem numbers and other identifying information associated with the assignment. If this is a team assignment, make sure that all team members' names are included.

An important part of working with graphics is the distinction between "raster" and "vector" graphics formats.

- Raster graphics formats are oriented towards producing colored pixels and subtly shaded areas, useful in photographs and artistic drawings. When images in these formats are resized or even just viewed on a device with a different resolution than the one in which it was drawn, the colors of adjacent pixels may be mixed (aliased) without noticeable degradation.

  Most of the diagrams that you are likely to produce in this class are line drawings mixed with text. Raster formats do not work well with these. When these are resized, the same mixing process results in fuzzy lines or may cause lines to disappear entirely.

  Examples of raster formats include `bmp`, `gif`, `jpeg`, and `png`. Use these with photograph-like images, not with line drawings. (If you absolutely must use one of these for line drawings, `jpeg` is probably the worst choice for this purpose.)

- Vector graphics formats are oriented towards the drawing of lines and geometric figures (including text). When images in these formats are resized, the lines are stretched by simply moving the coordinates of the end-points. Vector formats work very well with line drawings, remaining sharp and clear no matter how many times they are resized.

  They can be awkward with photographs or other images requiring subtle shading, but most vector formats will allow you to embed "boxes" of a common raster formats within a larger vector-format drawing.

  Common vector formats include `pdf`, `ps`, `eps`, `wmf`, and `emf`. Use these with line drawings.

  It can be a bit harder to find graphics programs that produce vector formats, and some word processors will let you paste vector images into a document but will not display it to you until you print.

  A tip: if you have a Windows drawing program that lets you move lines and shapes after placing them on the screen, it probably is using a vector format (`wmf` or `emf`) internally. In that case, if you copy-and-paste directly from your drawing program into your word processor, the graphic is probably being copied as a .wmf or .emf image even if the program does not allow you to save in those formats directly to a file.

C.3 Appendix: Protecting Your Files on the Unix System

The basic protection commands for Unix are covered in CS252. Here, however, I outline a simple setup that will make sure your files are protected from snooping without you're needing to worry about it every time you start a new assignment.

For the same of example, let's suppose that you are working on an assignment "asst1", with the instructor-provided files contained in the directory /home/cs330/Assignments/asst1.

1. If this is your first assignment for the course, create a course work area and protect it from access by anyone except yourself.

   ```
   mkdir ~/cs330
   chmod 700 ~/cs330
   ```

2. Create a working directory for this assignment:

   ```
   cd ~/cs330
   mkdir asst1 cd asst1
   ```

   You need not worry about protecting this directory, as it is inside a directory that only you can access.

3. Copy any files you need into this directory. E.g.,

   ```
   cd asst1
   cp /home/cs330/Assignments/asst1/* .
   ```

   It's important, by the way that you copy "*" and not "*.*". The latter will only copy files that have a '.' in their name. Many files do not. Most notably, the `makefile` I provide with most assignments does not.

   This **cp** command will not copy anything inside subdirectories of the assignment directory. Usually, that's OK because the only subdirectory is probably the `bin` directory where the sample solutions are kept, and you don't need a distinct copy of that.

4. You are now ready to start work.

   For most assignments, you can compile the code by just giving the command

   ```
   make
   ```

In future assignments, just make it a habit to do all of your assignments inside the protected `~/cs330` directory, and you won't have to worry about anyone else seeing your code.