EE 322C
Data Structures

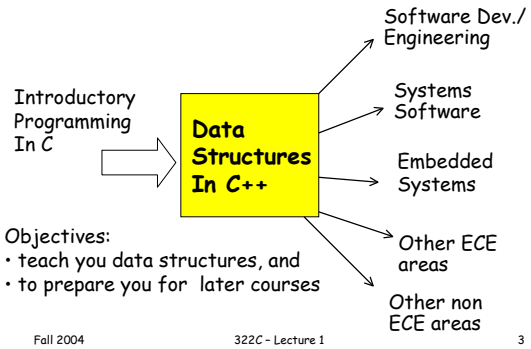
Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

Introductions

- **The Teaching Team**
  - Instructor – Dewayne E. Perry
    - Programmer, Designer and System Architect
    - Visiting faculty at CMU
    - SE Researcher at Bell Labs
    - Motorola Regents Chair in SE at UT Austin
  - TA – Matthew Hawthorne
    - MS in SE in the UT Option 3 Program
    - Significant experience in SE & C++
    - PhD Student working with me on SW Architecture
  - Grader – [don't know yet]

ECE322C in Context



Objectives:
- teach you data structures, and
- to prepare you for later courses

Goals for this class

- A practical understanding of a variety of common data structures
- A practical understanding of where they are applicable
- Knowledge of the basic constructs of the C++ and good programming style
- How to use C++ to create appropriate abstractions to solve programming problems
- A good understanding of basic software engineering principles

High Points of Syllabus

- Prerequisite, EE312- C Programming
  - If no prerequisite or not sure the prerequisite has been met, see me
- You are responsible for all materials presented in classes, whether you attend or not. Material presented in class is in addition to the notes.
- The purpose of the lecture notes is to help you listen in class.

Schedule Highlights

- Aug 25 is our first day of class, Dec 1 is our last day of class
- Lectures every Monday and Wednesday in CPE 2.210 from 5- 6:15
- Exams (3): Sep 27, Oct 27, and Dec 1
- Programming assignments (6-8) will come out throughout the semester
- No class Nov 24 (evening before Thanks-giving)

Assignments and Grading

- Assignments will be 6 - 8 programs.
  - Programs to be completed **independently** unless I state otherwise (we may try some pair programming)
- Three exams during semester.
- Pop quizzes at any time
  - all equal to 1 assignment
- Grades made up of:
  60 % exams
  40 % assignments/quizzes

Final Grade Criteria

Final Average	Letter Grade
90 – 100	A
80 – 89	B
70 – 79	C
60 – 69	D
0 - 59	F

Assignments and Grading

- Exam grades may be curved if warranted.
- Programs are submitted for grading via email system
- Assignments turned in late **will not** be accepted.
- Your program must run successfully on the ENS lab configuration
- Assignments are graded on a 20 point scale
  - Each assignment may have different criteria
  - Partial credit may be given
  - Correctness, style, performance, etc. will be scored

Syllabus

- All the remaining details of the course policies, rules, grading criteria, and procedures are in the syllabus document on the class web page
- Various C/C++ documents will also be available on the web page
- Will have the web page set up by next week

Questions?

Software Engineering and Programming

- Software Engineering (SE) is about
  - Building and evolving software systems
  - That solve practical problems in the world
  - Using appropriate and simplifying **abstractions**
- Programming is about
  - Finding the appropriate representations for
    - Processing
    - Data
  - Implementation details in a programming language (here C++)

Standard View of SE

- Basic SE life-cycle processes
  - Requirements
  - Architecture & Design
  - Construction
  - Deployment & Maintenance
- Integral to life-cycle processes
  - Documentation
  - Measurement & Evaluation (M&E)
  - Teamwork
  - Management of system objects
  - Evolution

Standard View of SE

- We will become acquainted with aspects of each of these
  - Requirements: the problem to solve
  - Design: the shape of the problem influencing the shape of the solution
  - Construction: integrating multiple pieces
  - Documentation: describing the solutions
  - M&E – various forms of analysis and testing
  - Teamwork: will do some projects in teams
  - Evolution: will evolve some projects

A Different View of SE

- Three elements in engineering SW systems
  - Theory
  - Experience
  - Process
- We will become acquainted with aspects of each of these
  - Will introduce theories for data structures
  - Will gain experience with them
  - Will instill good SE & programming practices

Wisdom from Fred Brooks

- Suggestion: read *Mythical Man-Month* often
- Essential Characteristics of SW Systems
  - Complexity – **our besetting problem**
    - *Software entities are more complex for their size that perhaps any other human construct*
    - Two kinds of complexity
      - Intricacy (may find some of this in some data structures)
      - Wealth of detail (probably not in this class)
  - Lack of Conformity
  - Changeability & Evolution
  - Invisibility and Implicitness

Wisdom from Fred Brooks

- Accidental Characteristics
  - Inadequate abstractions
    - Our main job as SEs is to find, create and evolve appropriate abstractions
  - Inadequate modes of expression
    - Depends on the languages we use
  - Language limitations – here C++
  - Resource limitations – time, PCs, cycles, etc
  - Inadequate support – tools, environments, etc

Managing Complexity

- Modularity
  - Divide and conquer
  - Break things up into manageable pieces
- Encapsulation
  - Localize similar things
  - Localize expected changes
- Abstraction
  - Functional: generalize and parameterize
  - Implementation:
    - Define simple interface
    - Hide implementation details

What is a Program?

- Algorithms + Data Structures = Program
- Data
  - Is information represented in a manner suitable for communication or analysis by humans or machines
  - A data structure is a systematic way of organizing, holding, and accessing computerized data
- An algorithm
  - Is a logical sequence of discrete steps that describes a complete solution to a given problem computable in a finite amount of time.
  - The key to packaging and time Vs. space tradeoff decisions

Structured Programming

- A disciplined style of programming where the
- Static structure mirrors the dynamic structure
  - Modularization and scoping of programs
  - Restricted set of control structures
  - Indentation of subordinate structures
- Control Structures
  - Sequence
  - Selection (if, Case, etc)
  - Iteration (for, while, etc)
- Data Structures
  - Tuples
  - Ordered elements
  - Unordered elements

Data Structures

- Base Types
  - int, float, char, bool, enum, pointer
- Tuples
  - struct
- Ordered types
  - string, array, vector, stack, queue, linked list, tree, graph, table, hash table
- Un-ordered types
  - sets, heaps

Questions about Data Structures

- When are the different data structures applicable or appropriate?
  - When do we use types, ordered or unordered structures?
  - What are the costs and benefits?
- How do you design new data types?
  - Open structures or abstract data types?
  - What operations are needed?
    - Eg, add, remove, access data
  - What else is needed?

Functional vs. Object-Oriented

- Read the problem statement and/or specification of the software you want to build.
  - Underline the verbs if you want to focus on procedural aspects,
  - Underline the nouns if you want to focus on the data aspects
- How do you decide which to emphasize in a system design ?
  - it depends on the application

Two SW Design Approaches

FUNCTIONAL DECOMPOSITION

Divides the problem into more easily handled subtasks, until the functional modules (subproblems) can be coded.

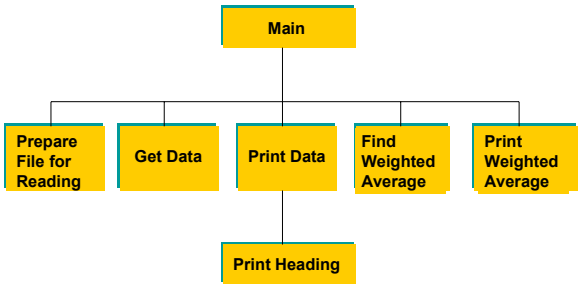
FOCUS ON: ***processes***

OBJECT-ORIENTED DESIGN

Identifies various objects composed of data and operations, that can be used together to solve the problem.

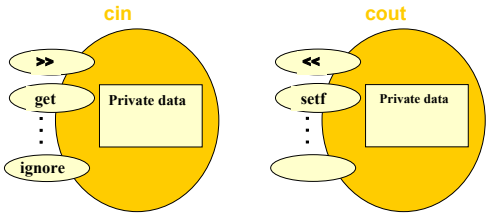
FOCUS ON: ***data objects***

Functional Design



Object-Oriented Design

A technique for developing a program in which the solution is expressed in terms of objects -- self- contained entities composed of data and operations on that data.



What is Software Engineering?

- A disciplined approach to the development of computer software systems that:
  - produces high quality software solutions (i.e. it works correctly, its reusable, modifiable, etc.),
  - are developed on time and within cost estimates,
  - uses technology that help to manage the size and complexity of the resulting software products.
  - applies to all types of software systems that are developed as products
  - uses general principles and domain specific approaches as well

What is System Software?

- Operating systems, compilers, linkers, loaders, middleware
- Network management tools
- Computer performance monitors
- Telecomm
- NOT end user applications, web aps, games, etc.
- Issues involved are very close to the machine: squeezing space, minimizing time, slicing resource utilization, etc.

What is Embedded Software?

- Inside a device
- Smart appliances
- Automotive, anti lock brakes
- Digitial signal processing
- System on a chip
- Issues involved are hard real time

Next Time

- We get started on C++

EE 322C
Data Structures

Lecture 2

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

Announcements

- http://www.ece.utexas.edu/courses/fall_04/ee322c-15515
  - Class web site is up
  - Basic C to C++ reading notes out
  - Syllabus is there
  - Class lectures will be there after
  - Assignments will be announced there

C++ References

- Books
  - *C++ Programming Language, 3rd Edition,* B. Stroustrup, Addison Wesley, 2000
- Online sources
  - *http://cppreference.com - syntax reference*
  - *http://www.cprogramming.com/tutorial.html*
  - *http://www.cplusplus.com*
  - Many others – search for them

Two SW Design Approaches

FUNCTIONAL DECOMPOSITION

Divides the problem into more easily handled subtasks, until the functional modules (subproblems) can be coded.

FOCUS ON: processes

OBJECT-ORIENTED DESIGN

Identifies various objects composed of data and operations, that can be used together to solve the problem.

FOCUS ON: data objects

Introduction to Problem Solving and Algorithms

- Look at algorithms first
- Fits the functional style of C
- Algorithms + data structures = programs

Algorithm

- **General** - a step by step procedure for solving some problem or accomplishing some goal (Webster's) - e.g. a recipe
- **Computer** - A logical sequence of discrete steps that describes a complete solution to a given problem that is computable in a finite amount of time.
  - "Find the largest prime number" is NOT amenable
  - Often appears at several levels of abstraction/detail
  - Algorithms operate on data structures from a functional viewpoint

Algorithm (cont.)

- A given problem may be solvable by a number of different algorithms. Its importance is crucial in designing a solution program.
- An algorithm may be transformed into a working program if its computable
- An algorithm will typically use levels of abstraction to make the solution clearer and implementation easier.

Algorithm (cont.)

- An algorithm may be represented in several ways:
  - **Pseudocode** - structured English language used to help design an algorithm (free form; e.g. recipe)
  - **Flowchart** - a graphical representation of an algorithm. It shows control and data flow.
  - **Formal languages** - outside the scope of this course
  - **Computer program** - eventually an algorithm is written in a programming language

Algorithms in General

How To Shampoo Your Hair

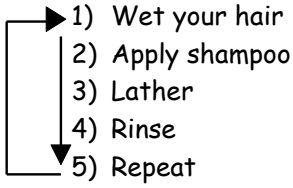
Follow these simple steps:

1) Wet your hair
2) Apply shampoo
3) Lather
4) Rinse
5) Repeat

Found on the back of a shampoo bottle - circa 1965

How To Shampoo Your Hair

Follow these simple steps:



Steps 1 - 4 depict a sequential flow of instructions
Step 5 introduces the notion of repetition/iteration of instructions

How To Shampoo Your Hair

Follow these simple steps:

0) If out of shampoo,
   then run out and buy some
1) Wet your hair
2) Apply shampoo
3) Lather
4) Rinse
5) Repeat

This is a decision making statement

How To Shampoo Your Hair

Follow these simple steps:

1) Wet your hair
2) Apply shampoo
3) Lather
4) Rinse
5) Repeat steps 1 - 4, if necessary

This is a bounded iteration statement

How To Shampoo Your Hair

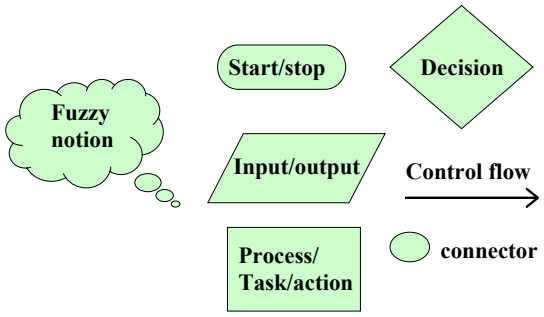
Follow these simple steps:

1) Wet your hair
2) Set the wash hair **counter** to 0
3) Repeat steps 3A - 3D while the value of wash hair **counter** is less than 3 (i.e. do it three times)
   A. Apply shampoo
   B. Lather
   C. Rinse
   D. Add 1 to the wash hair counter
4) Stop

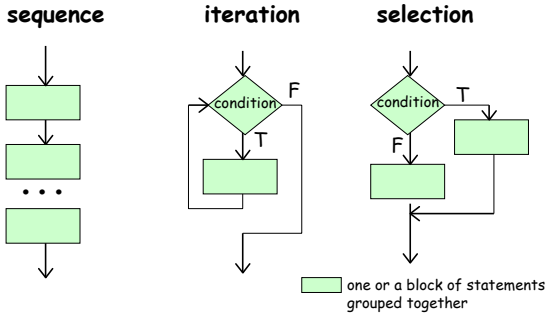
This is what pseudocode looks like

Algorithm Exercises

Flowchart Symbology 101
- Primitives -



Three Proper Logic Constructs



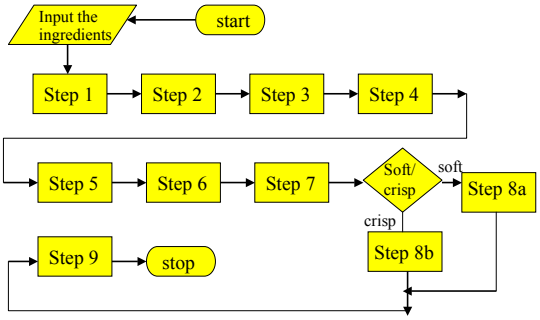
Three Proper Logic Constructs
- pseudocode style -

sequence	iteration	selection
1. …	Repeat below steps While condition is true	If condition is true then
2. …	a) … b) …	… else
3….	c) … end repeat	… end if

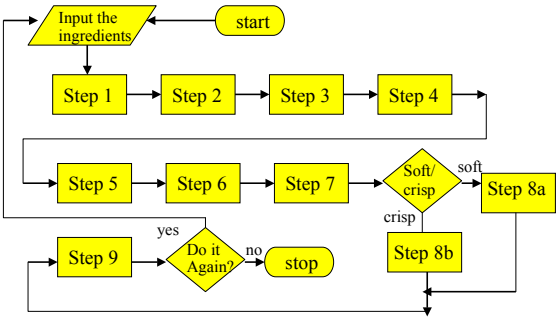
Use indentation heavily to show the static (ie, block) structure

Making Cookies

Judy's Chocolate Chip Cookies

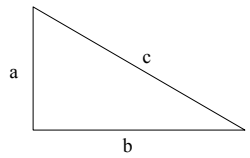


Judy's Chocolate Chip Cookies



Numerics - Example

- **Problem** - given the right triangle depicted in the figure below, and given values for the lengths of sides a and b, what is the length of side c?



Algorithm

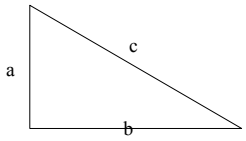
1. Input values for sides a and b
2. Compute the length of the hypotenuse c
3. Output the answer

Code Example

```
#include <iostream>
#include <cmath>
using namespace std;
int main ( )
{   // find the length of the hypotenuse c of a right triangle
    // with sides a,b,c
    double a, b, c;
    cout << "enter a value for side a" << endl;
    cin >> a;
    cout << "enter a value for side b" << endl;
    cin >> b;
    c =  sqrt (a*a + b*b);
    cout << "the value of c is " << c << endl;
    return 0;
}  // end of main
```

Numerics - Exercise

- **Problem** - given the right triangle depicted in the figure below, and given values for the lengths of sides a and b, what is the length of side c?



Do this for multiple given values of a and b

Algorithm

Repeat the following steps until ??

1. input values for sides a and b
2. compute the length of the hypotenuse c
3. output the answer

Code Example

```
#include <iostream>
#include <cmath>
using namespace std;
int main ()
  { // find the length of the hypotenuse c of a right triangle
    // with sides a,b,c
     double a, b, c;

          cout << "enter a value for side a" << endl;
          cin >> a;
          cout << "enter a value for side b" << endl;
          cin >> b;
          c =  sqrt (a*a + b*b);
          cout << "the value of c is " << c << endl;

     return 0;
     } // end of main
```


How To Solve Complex Programming Problems

Problem Solving Process

1) Understand the given problem
2) Devise a high-level plan to solve it
3) Elaborate the plan
4) Implement the plan
5) Revisit and revise the plan as necessary
6) Stop when the solution is correct

Problem Solving Process

1) Understand the given problem
2) Devise a high-level plan to solve it
3) Elaborate the plan
4) Implement the plan
5) Revisit and revise the plan as necessary
6) Stop when the solution is correct

Problem Solving Process

- Understand the given problem
  - Analyze the problem to determine: the goals, the givens,the requirements, the constraints
- Devise a high-level plan to solve it
  - Design the top-level system models (algorithm/data) to solve the problem- including new and reused sub-parts
- Elaborate the plan
  - Work out the details of the algorithm/data structures and their sub-parts until it is ready for coding in a programming language
- Implement the plan
  - Code, test and verify the program
- Revisit and revise the plan as necessary
  - Fix and/or modify the program as needed
- Stop when the solution is correct

Problem Analysis Process

1) Identify the inputs (e.g. the data given)
2) Identify the outputs (i.e. the desired results)
3) Identify the functions and features needed
4) Identify the data structures, variables and relationships between them
5) Identify additional requirements or constraints on the solution
6) Identify those processes that transform the input data into the output data
7) Identify pre-existing parts that will participate in the solution (e.g. library classes and methods)
8) Make assumptions and simplifications as necessary (document those)

Problem Analysis Tips

- Read the problem statement very carefully.
- Use the given analysis process to identify and understand the required inputs, desired outputs, etc.
- Parse the problem statement looking for the key concepts; use the **divide and conquer** approach
  - Noun phrases typically will denote potential data types and variables (and later: classes, objects)
  - Verb phrases will denote potential processes/functions/actions
  - Outputs can often be related to inputs by a transformation process (perhaps needing intermediate variables)
- **Work examples all the way through by hand**
- Seek clarification and more information as needed from the problem specifiers (the teaching team in this case)
- Create a high level sketch of the flow of the algorithm
- DO NOT start by trying to write C++ code!!

Divide and Conquer

- Programming is a complex task that is made simpler if you can break the big problem down into smaller subproblems
- These smaller pieces are components called modules - e.g. classes, methods, objects, subroutines, procedures or functions.
- Break down a problem into modules that each just do one thing using encapsulation and astraction. We will learn more about what makes a good module later.
- A key feature of a well written algorithm will be proper use of abstraction. This will provide an algorithm that:
  - is more understandable
  - is easier to implement and test
  - provides more reusable modules

Example Problem

Problem - You have $1,000 to invest in a Certificate of Deposit (CD) that earns 3% interest per year. Interest is compounded annually. How many years will it take to double your initial investment?

Example Problem

Problem - You have $1,000 to invest in a Certificate of Deposit (CD) that earns 3% interest per year. Interest is compounded annually. How many years will it take to double your initial investment?

Questions to you (the problem solver):

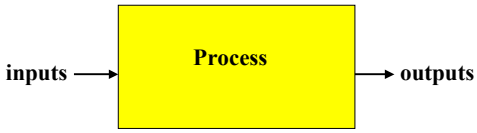
- Do you fully understand the problem statement? If not, what are your questions?
- Can you solve the problem by hand?
- Can you then teach the computer how to do it?
- Ask yourself, if I was a computer, what would I do first? then next, etc.
- What if the problem was changed slightly?
- Can you design a solution to solve this problem?

The Generic Program Design Usually Starts With
- *IPO* System Model -

To begin, represent the program as a closed, black-box system. Identify the inputs and outputs before you begin to describe and outline (plan) the steps inside the process box.

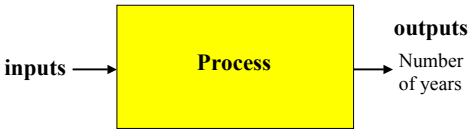
Example Problem

You have $1,000 to invest in a Certificate of Deposit (CD) that earns 3% interest per year. Interest is compounded annually. How many years will it take to double your initial investment?



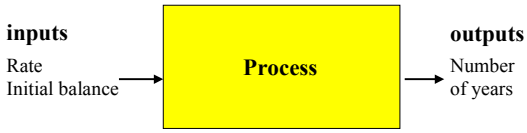
Example Problem

You have $1,000 to invest in a Certificate of Deposit (CD) that earns 3% interest per year. Interest is compounded annually. How many years will it take to double your initial investment?



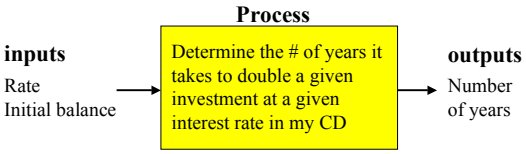
Example Problem

You have $1,000 to invest in a Certificate of Deposit (CD) that earns 3% interest per year. Interest is compounded annually. How many years will it take to double your initial investment?



Example Problem

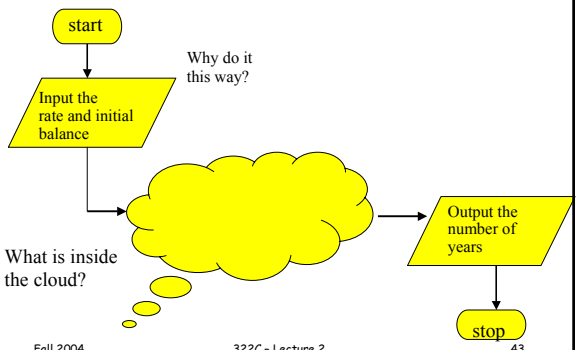
You have $1,000 to invest in a Certificate of Deposit (CD) that earns 3% interest per year. Interest is compounded annually. How many years will it take to double your initial investment?



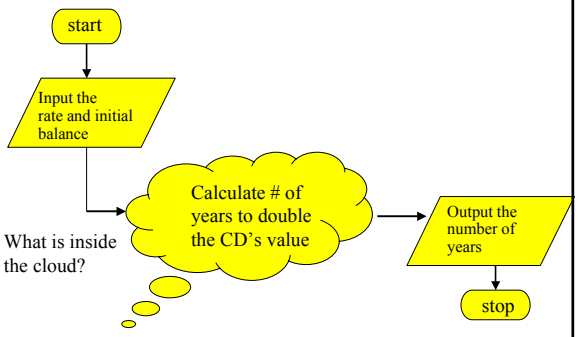
The Algorithm Design Process

- **Design the algorithm to solve the problem**
  1. Use top down decomposition/stepwise refinement to break a bigger process down into smaller pieces
  2. Continue until code can be written directly from the detailed algorithm
  3. Understand the primitives and piece parts you have to work with
  4. Locate relevant functions/classes/etc. in existing libraries
  5. Modify existing methods/classes where necessary
  6. Design new methods/classes where necessary

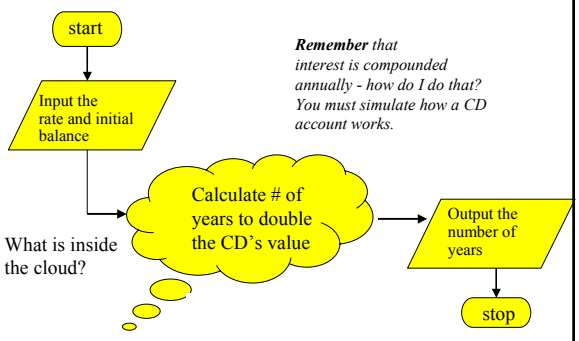
Algorithm - First Cut



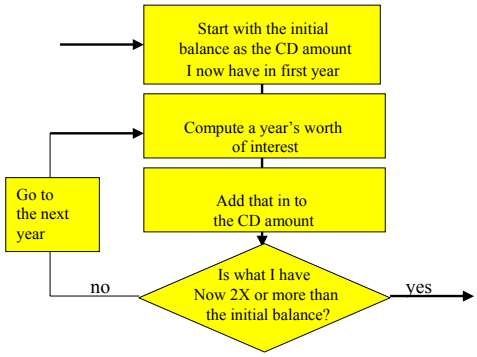
Algorithm - First Cut



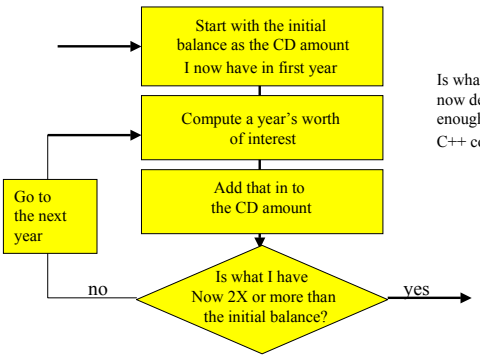
Algorithm - First Cut



Next Level Down



Next Level Down



Is what I have now detailed enough to write C++ code from?

Pseudocode Form

Input the rate and initial balance
Start with the initial balance as the CD amount in first year
Repeat the following until new balance >= 2X the initial balance
  Compute a year's worth of interest
  Add that in to the CD amount
  Go to the next year
Output the number of years

- Let's look at the program now

Programming Style and Documentation

- Appropriate Comments
- Naming Conventions
- Proper Indentation and Spacing Lines
- Block Styles
- For more information see the coding standards on the class web page

Appropriate Comments

- Comments are for the purpose of explaining and understanding of your program by humans
- Include a summary at the beginning of the program to explain what the program does, its key features, its supporting data structures, and any unique techniques it uses.
- Include your name, class section,assignment number, date, and a brief description at the beginning of the program.
- Also document the high level logic of your program at the key decision and iteration points.

Naming Conventions

Choose meaningful and descriptive identifier names that reflect the problem concepts.

- Variables and method names:
  - Use lowercase. If the name consists of several english words, separate them with underscores
  - For example, the variables radius and area, and the function compute_area.
- Symbolic Constants:
  - Capitalize all letters in constants. For example, the constant PI.
- Class names:
  - Capitalize the first letter of each word in the name. For example, the class name My_math_functions.

Proper Indentation,Spacing, Blocking

- Indentation
  - Indent three spaces to reflect each level of nesting
- Spacing
  - Use blank line to separate segments of the code.
- Block styles
  - Use next-line style for braces, and end of block comments. e.g.

```
class Test
{  int function1()
   {
      cout <<  "block style example";

      return 0;
   }  // end of function1
} // end of Test
```

EE 322C
Data Structures

Lecture 3

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

Announcements

- http://www.ece.utexas.edu/courses/fall_04/ee322c-15515
  - Class web site is up
  - Basic C to C++ reading notes out
  - Syllabus is there
  - Class lectures will be there after
  - Assignments will be announced there

Topics for Today

- OO thinking
- C++ starter stuff
- Quick review of strings

OO and C++

C++ supports object-oriented programming using abstraction, encapsulation, inheritance and polymorphism to provide great flexibility, modularity and reusability in developing software.

- We will eventually learn how to define, extend and work with **classes** and their associated **objects**
- But first we need to learn about how to think in an object oriented style
- Jargon - I will often use the term method to mean both functions and procedures

Classes and Objects "Introduction to Object Oriented (OO) Thinking"

"Objects are good" - Aristotle

Objects And Classes - in Two Worlds

Real World

- A **class** is an abstract grouping (or categorization) of related objects, containing members that have something in common (e.g. at least one attribute); it defines a general kind of or type of object (e.g. all cars, all students in this course)
- An **object** is a specific, identifiable thing; is a member of 1 or more classes, has attributes, exhibits behavior (e.g. my Toyota, your seat)

Objects And Classes - in Two Worlds

Real World

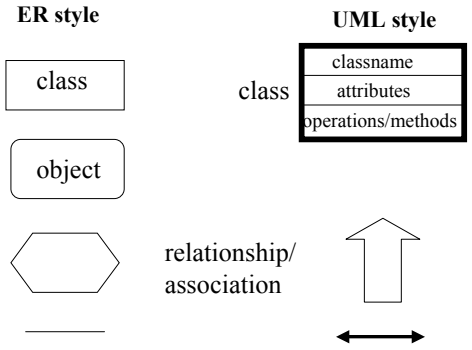
- A **class** is an abstract grouping (or categorization) of related objects, containing members that have something in common (e.g. at least one attribute); it defines a general kind of or type of object (e.g. all cars, all students in this course)
- An **object** is a specific, identifiable thing; is a member of 1 or more classes, has attributes, exhibits behavior (e.g. my Toyota, your seat)

C++ OO World

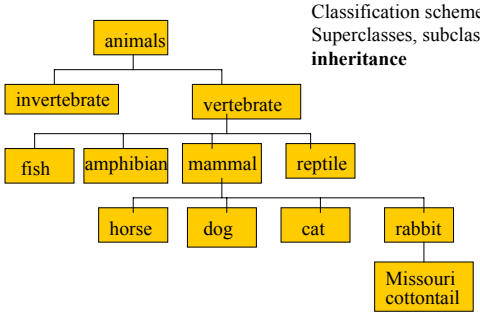
- A **class** is a container of functions, variables, etc.; and is a generalization of and a generator for all objects of that class
- An **object** is a member (or instance) of a class, it is a composite data structure with functions attached, and is a thing you can create and manipulate in your program. It has a state, and behavior.

For some programming problems it is easy to map between these two worlds - that is what OOP tries to do

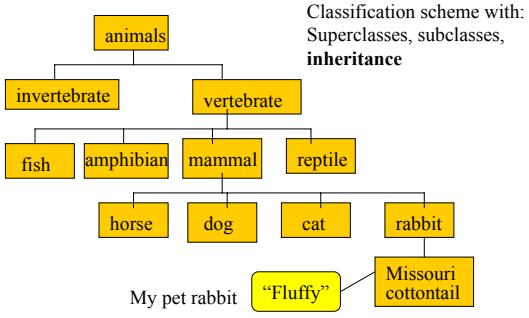
Object Model Graphics



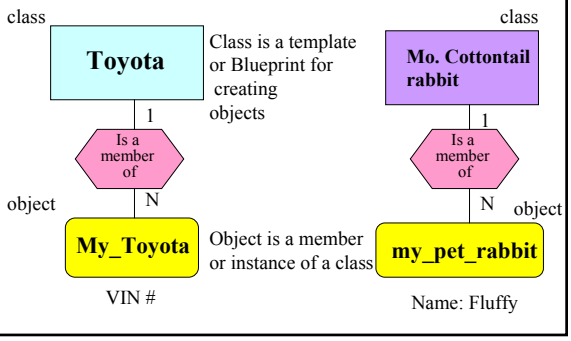
A Class Hierarchy



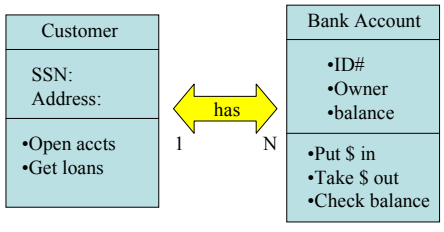
A Class Hierarchy



Membership Relationships Between Classes and Objects



UML Model Example

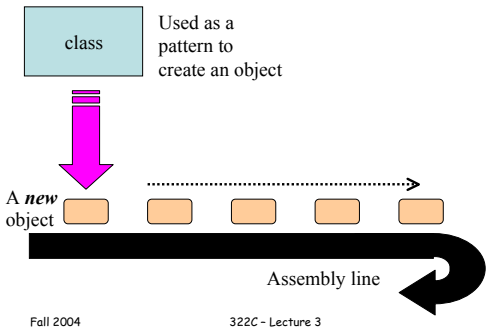


Purposes of C++ Classes

Classes serve the following purposes:

1. Creates a new programmer defined data type
2. A class is like a factory used to create (or construct) objects of that data type.
3. Specifies the functions (methods) you can use for objects that belong to that class.
4. Defines the common attributes of all objects in the class
5. A class defines (and encapsulates) the implementation details. E.g. data fields and code for methods
6. There are public parts and private parts

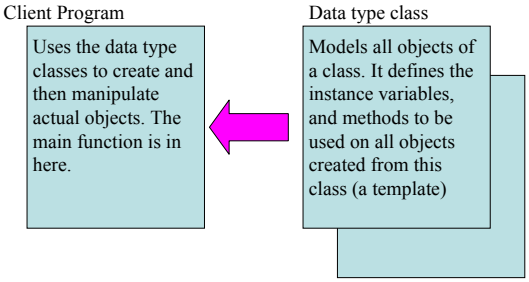
Class as Generator of Objects



Example Class Definition

```
class Bank_account  // this is a class definition,
  NOT an object!!
{// function (operations) definitions
  public: // means theses are known everywhere
     void deposit(double amount)
     {//  body left out for now
     }
     void withdraw(double amount)
     {//  body left out for now
     }
     double get_balance( )
     {//  body left out for now
     }
  // member attributes (state) variables definition
  private: // means it is known inside the class only
     int account_number;
     string owners_name;
     double balance;
}
```


OO Style Program



C++ Starter Stuff

Getting Started With C++

```
/* a program that does nothing
   and just returns a zero */
int main()
{
  // body of the program here
  return 0; //return an integer value of zero
}
```

- A `main` is required for all C++ programs
- Unlike C,it must have a return type of `int`, and should return a zero to indicate normal execution
- The body of `main` and all other functions should be enclosed within `{ }`.
- The statements in the body can (but should not) be written in a free-form where each statement is terminated by a ';' (semicolon).
- The `main` can have parameters to allow access to command line arguments (more later)
- The source code files can have various prefixes, e.g..cpp to indicate a C++ source program to the compiler
- Multi-line comments use '/*' and '*/' and single line comments use '//'

Header Files

```
#include <iostream>
#include "myincludes.h"
```

- Prototypes and declarations are kept in header files
- `#include` pre-processor directive inserts the file contents. No semicolon is needed.
- Header files of standard system and library functions are enclosed within "< >" and the compiler looks for those in the predefined paths.
- Header files within double quotes are searched for relative to the current directory or the user directory defined in the IDE
- The system header files do not have the suffix to allow different compilers to have their own suffix types. In C, the suffix was required.

Namespaces

- Namespaces allow the same names to be reused under a different qualifying namespace contexts
- All of the standard C++ libraries are wrapped inside of the `std` namespace, which has to be explicitly declared, e.g.

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

- Otherwise each use of a library function would need to have the namespace prefix. For example:

```
cout << "hello world\n";
```

would have to be written as:

```
std::cout << "hello world\n";
```

- The names from the namespace are visible only within the scope of the `using` directive (file, function, etc.)
- Names can be used selectively from a namespace

```
using namespace std::cout;
```

- You can define your own namespaces in C++

Hello World

```
#include <iostream>
using namespace std;
int main ()
{
   cout << "Hello World" << endl;
   // or   cout << "Hello World\n";
   return 0;
}
```

- `cout` or "console output" stream is used to output from a C++ program. `cin` is the "console input" stream.
- `endl` writes a newline ('\n') and flushes the output buffer. `ends` only flushes the buffer. Explicit flush can be done by `cout.flush();`
- Any number of `<<` operators can be used in one statement
- Some common functions are: `fill(char), precision(int), width(int)`
- Dot operator is used for calling a member function

Stream I/O Manipulators

- Look at http://www.cppreference.com - under C++ I/O for the basics and differences
- Some common stream manipulators are: `dec, endl, ends (output a null), flush, hex, oct, resetflags(long), setfill(char), setiosflags(long), setprecision(int), setw(int), ws`
- Other "state" modifiers or flags for `cout` (and other streams, fstream, etc) that can be used with `cout.setf(ios::flags)` `skipws, left, right, internal(padding for sign), dec, oct, hex, showbase, showpoint (trailing zeros), uppercase, showpos (show sign), scientific, unitbuf (flush after each op), stdio (flush after each char), fixed (dddd.dd)`
- The effect of manipulators/state modifier flags remain effective until they are reset

Variables

```
#include <iostream>
using namespace std;
int main ()
{   cout << "Enter a number" << endl;
   { int number=0;
     cin >> number;
     cout <<"you entered " << number << endl;
   }
   // cout << "number entered = " << number<<endl;
   // will fail on compile since it's out of scope
   return 0;
}
```

- Variables can be defined anywhere and are valid within the scope of the block in which they are defined
- They should be defined right before their first use, but within the scope of all their uses

Built-in Data Types and Operators

- A new data type `bool` is introduced that has two values, `true` and `false`, which are set to 1 and 0, respectively.
  - E.g. `bool married = true;`
- All the usual C data types such as *`float, double, int, char, void, etc`* as well as specifiers such as `short, long, unsigned and signed`.
- All the usual mathematical, logical, bitwise, sizeof, casting, etc. operators are supported. The logical operators produce `bool` values.
- Explicit casting syntax is introduced in C++ in addition to using C style cast operation and implicit conversions.
  - `static_cast<type>(var)` `//regular cast for type conversion`
  - `const_cast<type>(var)` `//const/volatile to non const and //non-volatile pointers`
  - `reinterpret_cast` `//for complete disregard of a type`
  - `dynamic_cast` `//for objects`

Quick Review of strings

- What are strings? Two answers:
  - C style string is an array of chars terminated by the NULL char
  - C++ style string is a composite object from the new data type named string (a class definition); which also contains some useful operations that can be performed on strings
- C++ strings are better than C style strings
- Use the C++ standard library `string`
  - `#include <string>`

About strings

- An **object** is composed of values, and has associated methods that can operate on it (more later)
- A string literal is 0 or more characters enclosed in double quotes.
  "" is the empty or null string.
- For example `"Spanish Inquisition"`
  - The quotes are not a part of the string, they delineate the string.
  - If the string is output the double quotes do not appear
  - `cout << "Spanish Inquisition";`
  - **Spanish Inquisition** shows up on the computer screen
- String variables are declared and may be initialized, such as:

```
string name = "Ned Logan";
```


How characters are stored in strings

- Each character(char) in a string is in a sequential position.
- Each position has a number starting with position 0

Position #	0	1	2	3	4	5	6	7	8
string contents	N	e	d		L	o	g	a	n

- The number above each character specifies its position number (sometimes called its index number) in the sequence

What do we do with strings?

- Input and output them
- Make a bigger string out of little ones
- Break big strings into smaller ones
- Do comparisons (like in chars)
- Extremely useful in any application that manipulates text (e.g. translators, word processors, language puzzles, etc.)
- Useful methods for manipulating strings can be found at http://www.cppreference.com - under C++ strings

Concatenation

- The plus operator, +, has a special meaning for string objects
- + is used to *concatenate* two or more strings
  - Means append together, end-to-end, to form a new string, e.g.

```
string age = "9";
string s = "He is " + age + " years old";
cout << s; // He is 9 years old
```

- Concatenation can also be used with strings and other data types. - automatic conversion to string is done

```
int age = 9;
string s = "He is " + age + " years old";
cout << s;  // He is 9 years old
```


Other Simple Examples

- **Input/output of strings**

```
#include <string>
        . . .
cout << "please enter your name: ";
string s;
cin >> s; // needed storage is then allocated
cout << "Hello " << s + "!" << endl;
    . . .
```

- **Simple manipulations**

```
string str1 = "test"; // initializes str1
string str2;          // null string
str2 = "ing";         // allocates storage for "ing";
str1 = str1+str2;     // concats the two strings as "testing"
                // frees previous storage for str1
               // and reallocates for concat string
cout << str1[3];      // fourth character of string→ 't'
str1=str1.substr(0,4);//str1="test" from str1="testing"
if (str1 < s ) then i++;
```

String functions you should know

Given the declarations:

- `string str, string1, search_string;`
- `int index, start_index, number_of_chars;`

Here are examples of the most commonly used functions:

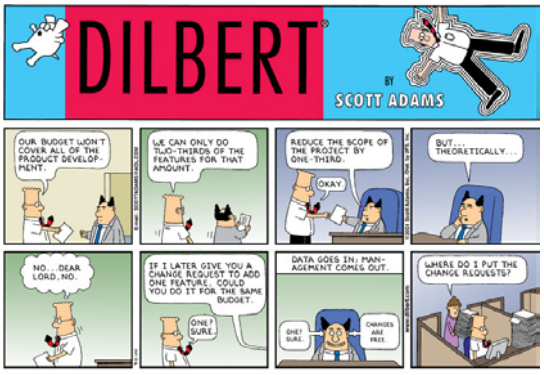
- `str.length ( )`
- `getline (cin, string1)`
- `str.find (search_string, start_index)`
- `str.substr (start_index, number_of_chars)`
- `str.insert (index, string1)`
- `str.erase (index, number_of_chars)`

EE 322C
Data Structures

Lecture 4

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm



Lecture 4 Announcements

- Homework Assignment on Web
  - Details on submission etc
- Coding standards available on web
- Topics of the day
  - More on C to C++ basics
  - Building your own data types
  - Booleans
  - Functions (time permitting)

What's Wrong with C ?

- It's a middle level language
- Its not strongly typed
- Its not machine independent
- It has little run time error checking
- It allows bit twiddling - OK for systems programming
- Not block structured, but function based
- Its not OO – no mechanisms for abstract data types (ATDs)
- It places few restrictions on programmers
- Good for embedded systems and system aps
- It has no graphical programming capability
- Lots of other little nits that are annoying

C/C++ Language Basics

- History, background, context, general programming concepts, program structure
- Expressions - basic data types, vars, operators, cont, expression evaluation rules
- Statements - selection, iteration, case, blocks, jumps
- Arrays - 1D, 2D, 3D?
- Pointers - vars, operators, with arrays, indirection, dynamic allocation of memory
- Functions - definition, calling, scope, args, parameters, return, prototypes, recursion?, std fcn library, overloading?
- Structs, unions, enums
- Console I/O - C style, C++ style
- File I/O - ?
- Preprocessor commands - #include

Keywords in C++ (63 vs 32)

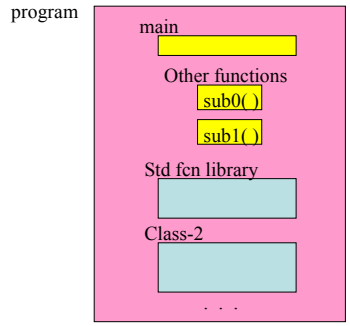
asm
auto
bool
break
case
catch
char
class
const
const_cast
continue
default
delete
do
double
dynamic_cast
else
enum
explicit
export
extern
false
float
for
friend
goto
if
inline
int
long
mutable
namespace
new
operator
private
protected
public
register
reinterpret_cast
return
short
size
sizeof
static
static_cast
struct
switch
template
this
throw
true
try
typedef
typeid
typename
union
unsigned
using
virtual
void
volatile
wchar_t
while

The Order of A C++ Program

#include statements
Base-class declarations
Derived-class declarations
Non-class function prototypes
Global variables declarations
int main ( )
{ // the body of your main program
}
Non-class function definitions

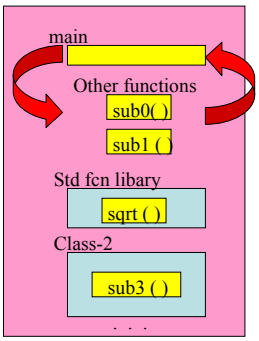
- **In most projects the class declarations will be in the standard class library or will be put into a header file and included with your program**

The block structure of a C++ Program



The block structure of a C++ Program

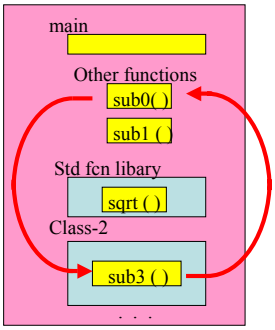
A function either returns a value through its name or returns no value (void)



A function CALLS another function by its name( ) to do some task

The block structure of a C++ Program

A function either returns a value through its name or returns no value (void)



A function CALLS another function by its name( ) to do some task

Calling a Function

What function? Where? How?

- str.translate ("Hello Class");

Qualifiers of the function call. In this case **str** is the name of a string variable

Function name - A function is called or invoked by using its correct name. A function is like a preexisting small program found in your code file or the C++ libraries.

Argument(s) are values that are sent to the called function

In this case the function translate requires a string parameter (e.g. "Hello, Herb") as the value to be operated on

Teaser - Is this a true function or a procedure being called ?

Function Prototypes

- All functions must be declared (signatures only) before using. This allows for compatibility and type checking during compilation. E.g.
  - int foo1(void);
  - double foo2 (int, int, bool);
- *void* is optional in C++ if a function takes no parameters.
- Every function must have a prototype with the parameter list. However, only parameter types need to be declared (names are optional).
- In C++, the functions must return a value as declared.
- All default parameters must appear to the right.
  - int foo4(int i=0, char); // wrong -- int i=0 should be on the right side of char
- an example

```
void square_it (double);  // the prototype
int main ( )
{  int x = 10;
   square_it (x); // what's the problem?
   return 0;
}
```

Build Your Own Data Types

The below are built on top of the C++ built-in data types: int, float, double, char, void, array, pointer

- **struct** - aggregate variables under one name
- **bit-field** - struct with bit level access
- **union** - two or more types for same memory
- **enum** - list of int constants
- **typedef** - alias for another type

Add in two new built-in data types: bool, wchar_t

The following are advanced capabilities for creating new data types

- **class** - encapsulates code and data as a logical abstraction
- **template** - used to create generic classes and functions

The `boolean` Data Type

- A boolean variable is used to hold truth values: either **true** or **false** (which are reserved words that stand for boolean literal values).
- Boolean variables are declared and then used as logic flags, switches, etc..

For example:

```
boolean lights_on = true;
       // Later in the program
lights_on = false;

boolean married = true;
If (married) cout << "hitched";
```


Boolean Expressions

- Boolean expressions are used in conditional statement clauses (if, while, for, do), OR on the right hand side of an assignment to a boolean variable.
- A Boolean Expression is a formula that evaluates to either true or false, e.g.

```
int A,B,C;
// assume that A,B, and C get values from input
boolean flag  =  A < = B + C;
// what's the value of flag?
```

- Boolean expressions can get complex, just like any formula

Relational Operators

Conditions are often mathematical comparisons using these

Mathematics		C++ Language	
Symbol	Name	Symbols	C++ Example
=	equal	= =	balance = = 0
≠	not equal	!=	answer != 332
>	greater than	>	expenses > income
≥	greater than or equal	>=	points >= 60
<	less than	<	pressure < max
≤	less than or equal	<=	expenses <= income

Logical Operators

- There are three *boolean logical operators* that allow us to create more complex boolean expressions that contain many sub-conditions connected by these
- && Logical AND, all conditions must be true for the whole boolean expression to be true
  - `if(status == SINGLE && income < 21450)`

Logical Operators

There are three *boolean logical operators* that allow us to create more complex boolean expressions that contain many sub-conditions connected by these

- && Logical AND, all conditions must be true for the whole boolean expression to be true
  - `if(status == SINGLE && income < 21450)`
- || Logical OR, if one of the conditions is true the whole boolean expression is true

```
- if(month == APRIL || month == JUNE ||
  month == SEPTEMBER || month == NOVEMBER)
     daysInMonth = 30;
```

Logical Operators

- There are three *boolean logical operators* that allow us to create more complex boolean expressions that contain many sub-conditions connected by these
- && Logical AND, all conditions must be true for the whole boolean expression to be true
  - `if(status == SINGLE && income < 21450)`
- || Logical OR, if one of the conditions is true the whole boolean expression is true
  - `if(month == APRIL || month == JUNE || month == SEPTEMBER || month == NOVEMBER)`
    - `daysInMonth = 30;`
- ! Logical NOT, gives the opposite of the condition

Boolean Logic

- If A and B are conditions that are each either true or false (e.g. A is x <= 10, B is y > 5) then the following are the *truth tables* for our three boolean logical operators:

A	B	A || B	A && B
true	true	true	true
true	false	true	false
false	true	true	false
false	false	false	false

A	! A
true	false
false	true

Boolean data types (more)

- boolean variables may be used wherever a boolean expression may be used.
- boolean variables may be assigned the value of a boolean expression, e.g.

```
int hours, year; //assume these have values
const int SENIOR = 4;  //declaring a constant
boolean happy = false;
happy = (hours <= 12) && (year != SENIOR);
if (happy)
{ // what does such a person do?
}
```

- Boolean variables can be used to simplify the conditional logic in your program

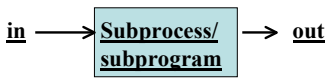
Loops — two basic patterns

Count-controlled loop:
- Number of iterations is determined before the loop starts.
- Counts each iteration using a counter variable.
- Stops when the desired number of iterations has been performed.

Event-controlled loop:
- Before each iteration, checks to see whether some event has occurred.
- Continues until that event occurs.
- Number of iterations not known beforehand.
- The event signal is in the condition to be tested; and may change during an iteration. A boolean variable is often used to flag the signal, e.g.

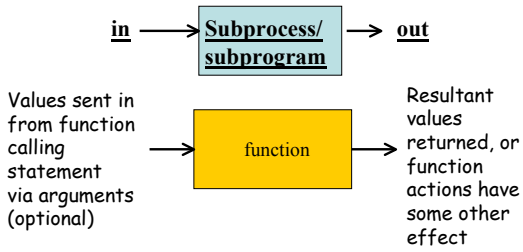
```
while(notSatisfied) //a boolean variable
{
  // do other stuff
}
```


A Function as a Module

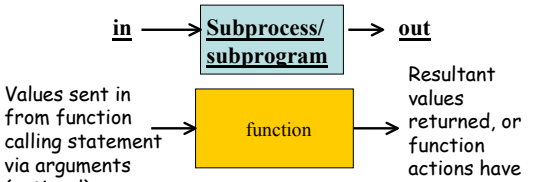


Defining and using functions are a primary way of modularizing the procedural aspects of your programs in C++ (or any language)

A Function as a Module



A Function as a Module



As in, for example:

```
double y = sqrt (x);
getline (cin, string1 );
string2 = str.substr (startIndex, endSpot)
```


String Reverser example

```
int i =0;
            char ch;
            string  phrase, reversed;
            getline (cin, phrase );
            reversed = "";
            while (i  <  phrase.length())
  {   ch = phrase[i];
            reversed = ch + reversed;
            i = i + 1;
  }
  cout <<  reversed;
```


String Reverser Example

```
int i =0;
            char ch;
            string  phrase, reversed;
            getline (cin, phrase );
            reversed = "";
            while (i  <  phrase.length())
   {    ch = phrase[i];
              reversed = ch + reversed;
              i = i + 1;
   }
   cout <<  reversed;
```

- Now suppose that we need to reverse many different strings in several places in our main function - do we really want to cut and paste this code into many places or should we turn this code into a "module" (function) that can be called from many places as we have already done with the library functions that we use?

The String Reverser Function

```
string  reverseFunction (string phrase)
  // function signature/ header
{ int i =0; char ch;
  string reversed;
  reversed = "";
  while (i  <  phrase.length())
  {   ch = phrase[i];
        // concatenate ch onto the front-end of reversed
      reversed = ch + reversed;
      i = i + 1;
  }
  return reversed;
      // means send back the answer to the call
}
```

This function can be called from within your main(or other) function in the following way:

```
   string  result = reverseFunction ("actual text");
```


Call by Value or Reference

```
void foo1(int)
void foo2(int*)
int main()
{  int value=1;
   foo1(value);
   cout << value <<endl;   //outputs 1;
   foo2(value);
   cout << value <<endl;   //outputs 2;
   return 0;
}
void foo1(int a)
{ a++;}       //e.g. call by value
void foo2(int &a)
{ a++;}   // e.g. call by reference
```


Structures 101

- **Aggregation of variables (elements) under one name**
- **Declaration forms a template that can be used to create structure objects**

```
struct address
{ string  house_number;
   string street_name;
   string city;
   string state;
   int zip_code;
};
```

- **Use the structure definition to create variables and objects, e.g.**

```
address my_address;
my_address.city = "Austin";
address your_address = my_address;
```

- **We can create arrays of structs too**

EE 322C
Data Structures

Lecture 5

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

Rerun Last Part of Lecture 4

String Reverser Example

```
int i =0;
char ch;
string  phrase, reversed;
getline (cin, phrase );
      reversed = "";
while (i  <  phrase.length())
{
    ch = phrase[i];
    reversed = ch + reversed; // ch[n] … ch[1]+ ch[0]
    i = i + 1;
}
cout <<  reversed;
```

- Now suppose that we need to reverse many different strings in several places in our main function - do we really want to cut and paste this code into many places or should we turn this code into a "module" (function) that can be called from many places as we have already done with the library functions that we use?

The String Reverser Function

```
String reverseFunction(string phrase)//function signature
{
  int i =0; char ch;
  string reversed;
  reversed = "";
  while (i  <  phrase.length())
  {
      ch = phrase[i];
      reversed = ch + reversed;
      i = i + 1;
  }
  return reversed; // send back the answer to the call
}
```

This function can be called from within your main(or other) function in the following way:

```
string  result = reverseFunction ("actual text");
```


Call by Value or Reference

```
void foo1(int)
void foo2(int*)
int main()
{
   int value=1;
   foo1(value);
   cout << value <<endl;   //outputs 1;
   foo2(value);
   cout << value <<endl;   //outputs 2;
   return 0;
}
void foo1(int a)
{ a++; }   //e.g. call by value
void foo2(int &a)
{ a++; }   // e.g. call by reference
```


Lecture 5 Announcements

- Change: Assignment 1 due next Monday
- Topics of the day
  - Overloaded functions
  - structs
  - Bit fields
  - Unions
  - Enumerations
  - Typedef
  - Classes and objects

Overloaded Function Names

- Two or more functions can share the same name as long as their parameters are different
- This makes the function call context sensitive

```
#include <iostream>
using namespace std;
// abs is overloaded three ways
int abs(int i);
double abs(double d);
long abs(long l);
int main()
{ cout << abs(-10) << endl;
  cout << abs(-11.0) << endl;
  cout << abs(-9L) << endl;
  return 0;
}
int abs(int i)
{ cout << "Using integer abs()\n";
  if ( i<0) return -i;  else return i;
}
double abs(double d)
{ cout << "Using double abs()\n";
   if ( d<0) return -d; else return d;
}
long abs(long l)
{ cout << "Using long abs()\n";
 if ( l<0) return -l; else return l;
}
```


Build Your Own Data Types

The below are built on top of the C++ builtin data types: int, float, double, char, void, array, pointer

- **–struct** - aggregate variables under one name
- **–bit-field** - struct with bit level access
- **–union** - two or more types for same memory
- **–enum** - list of int constants
- **–typedef** - alias for another type

Add in two new builtin data types: bool, wchar_t

The following are advanced capabilities for creating new data types

- **–class** - encapsulates code and data as a logical abstraction
- **–template** - used to create generic classes and functions

Structures

- **Aggregation of variables (elements) under one name**
- **Declaration forms a template that can be used to create structure objects**

```
struct address
{
  string  house_number;
  string street_name;
  string city;
  string state;
  int zip_code;
};
```

- **Use the structure definition to create variables and objects, e.g.**

```
address my_address;
my_address.city = "Austin";
address your_address = my_address;
```

- **We can create arrays of structs too**

```
address  my_neighborhood [100];
my_neighborhood [3].state = "TX";
```


typedef

- Provide an alias for a data type
- Used to make code easier to read or easier to port or create a more familiar language (as in my thinking in Java) e.g.
- `typedef bool boolean;`

```
// I can now use the word boolean as well as
// bool when I declare a boolean variable
boolean married = true; // or
bool married = true;
```


Bit Fields

- A special type of struct
- Allows access to single bits in memory
  - E.g. for device drivers and encryption methods
- Beware of machine dependencies and restrictions

```
struct device_status_byte
// definition of a status byte from a comm.port
{
  unsigned delta_cts: 1;
  unsigned delta_dsr: 1;
  unsigned tr_edge:   1;
  unsigned delta_rec: 1;
  unsigned cts:       1;
  unsigned dsr:       1;
  unsigned ring:      1;
  unsigned rec_line:  1;
}
```


Bit Fields

- Example

```
// example usage
device_status_byte  status;
status = get_port_status();
if(status.cts)  cout << "clear
to send";
if(status.dsr)  cout << "data
ready";
status.ring = 0;
. . .
```

Unions

- Logical: defines variant interpretation of data
- Physical: memory locations shared by different variables, usually of different types, at different times, e.g.

```
union pw
{
   short int i;
   char ch[2];
};
. . .
   union pw word;          // create a pw object
   const int MAGIC_NUMBER = 19813;  // its "Me"
   word.i = MAGIC_NUMBER ; // set it as an integer
   cout << word.ch[0];     // write first char
   cout << word.ch[1];     // write second char
   cout << sizeof (word);  // sizeof returns the # bytes of any
                           // variable
```


Enumerations

- An enumeration type lists a named set of values
- You may also specify the integer values that represent the legal values for the named values (or use the compiler assigned defaults)

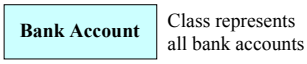
```
enum US_coin_value
{ PENNY=1, NICKEL=5, DIME=10, QUARTER=25, HALF_DOLLAR=50, DOLLAR=100};
US_coin_value  money;  // declare a variable of that type
money = dime;
if(money == quarter) cout << "Money is a quarter.\n";
switch(money)
{
      case  PENNY:   cout << "penny";     break;
      case  NICKEL:  cout << "nickel";    break;
      case  DIME:    cout << "dime";      break;
      case  QUARTER: cout << "quarter";   break;
      case  HALF_DOLLAR: cout << "half_dollar";  break;
      case  DOLLAR:  cout << "dollar";    break;
      default:       cout << "Money is not a legitimate coin value";
}
```


Purposes of C++ Classes

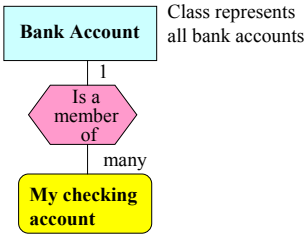
Classes serve the following purposes:

1. Creates a new programmer defined data type
2. A class is like a factory used to create (or construct) objects of that data type.
3. Specifies the functions you can use for objects that belong to that class.
4. Defines the common attributes of all objects in the class
5. A class defines (and sometimes hides the) implementation details. E.g. data fields and code for functions

Modeling Bank Accounts (OO)

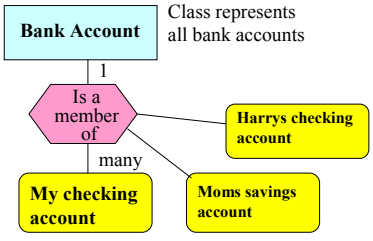


Modeling Bank Accounts (OO)



An object is a specific member of the class

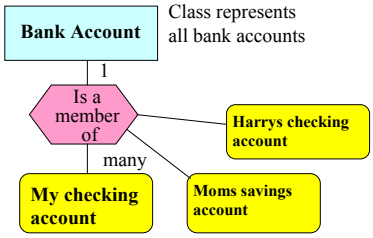
Modeling Bank Accounts (OO)



An object is a specific member of the class

Modeling Bank Accounts (OO)

What are the common attributes of and behaviors associated with all bank accounts?



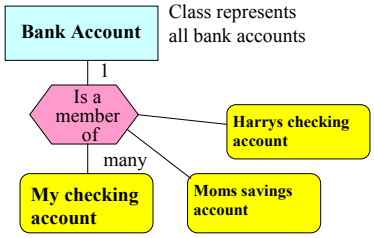
An object is a specific member of the class

Modeling Bank Accounts (OO)

What are the common attributes of and behaviors associated with all bank accounts?

Attributes:
- Account #
- Owners name
- Balance

Behaviors
- Deposit $
- Withdraw $
- Check current balance



An object is a specific member of the class

Object State

- Bank account attributes
  - Account number
  - Account owner's name
  - Current balance
- Instance/member Variables
  - accountNumber
  - ownersName
  - balance

Object Behaviour

- Bank account operations
  - deposit money
  - withdraw money
  - get the current balance
- Functions (non static)
  - deposit
  - withdraw
  - getBalance

Abstract Data Type

An abstract data type (ADT) is a high level description of a new data type to be implemented.

The kind of objects and their common attributes and operations are described, in general.

E.g. This is a definition of an abstract class of related bank account objects.

Abstract Data Type

Bank Account: a bank account is …
Common Attributes:
balance - the current value in $$ in this account
owner - the name of the person that this account belongs to
account_ID - a unique number assigned by the bank that identifies this account
Common Operations:
deposit - updates the current balance by adding in a given amount
post condition: the new balance is increased by the amount of $$
withdraw - updates the current balance by subtracting the given amount
post condition: the new balance is decreased by the given amount of $$
pre condition: the current balance must have at least the given amount in it
get_balance - returns the value of the current balance of the account

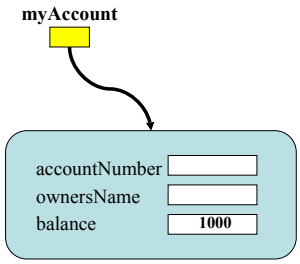
C++ Class Definition Example

```
class BankAccount  // this is a class, NOT an object!!
{
  // attributes - instance (state) variables definition
  private:
     int account_number;
     string owners_name;
     double balance;
  public:
     // function protoype definitions
     void deposit(double amount);
     void withdraw(double amount);
     double get_balance( );
```


Creating a new Object of a Class

- To declare an object variable of that type we say:
  `BankAccount myAccount;`
- To dynamically allocate space for a new object of that class we say:
  `new BankAccount( )`
- To save an object reference in an object variable
  `myaccount = new BankAccount( );`
- To apply a function to an object of that class
  `myAccount.deposit (1000);`
- To reference an object's data items we say:
  `myAccount.ownersName // oops`

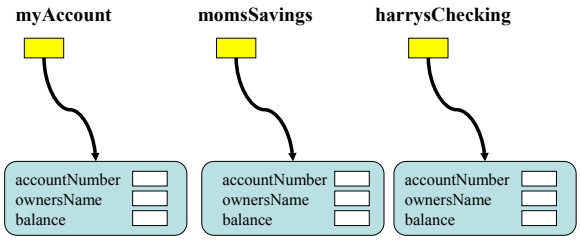
In C++'s Object Memory Bank



Constructing Several Objects of the Same Class

```
// e.g. in the main function
. . .
BankAccount myAccount      = new BankAccount();
BankAccount momsSavings    = new BankAccount();
BankAccount harrysChecking = new BankAccount();
```


Bank Account Objects



In C++'s Object Memory Bank

The Whole Class Definition

```
class BankAccount
{ // as before . . . plus constructor in public area
  BankAccount :: BankAccount (int, string, double);
}
// function bodies are defined here
void BankAccount :: deposit(double amount)
  {  balance = balance + amount; }
void BankAccount :: withdraw(double amount)
  {  balance = balance - amount; }
double BankAccount :: getBalance()
  {  return balance;  }
BankAccount :: BankAccount
  (int account, string name, double initialBalance)
{
  accountNumber = account;
  ownerName     = name;
  balance       = initialBalance;
}
```

Explicit Constructor function

Driver Program

Driver (client) Program

Uses the `BankAccount` class to create and then manipulate bank account objects.
The main function is in here.

`BankAccount` class

Models bank accounts. It
defines the instance variables, and functions to be used on all objects created from this class (a template)

`BankAccount` Driver Program

```
#include (BankAccount.h)
int main ( )
{
     BankAccount myAccount = new BankAccount(1, "me",10000);
     const  double INTEREST_RATE = 5;
     int years = 10;
     double interest;
  // compute and add in interest for 10 years
     for (int i =1; i  <=  years;  i++)
     {
       interest = myAccount.getBalance() * INTEREST_RATE/100;
       myAccount.deposit(interest);
       cout <<  "Balance after year " +  i  + " is $"
                + myAccount.getBalance( ) << endl;
     }
   return 0;
}
```


ADT/OO Tips

- Make data private
- Make functions public
- Separate the definition of a class from the use of that class
- This defines an API
- Static variables and functions that operate on the class as a whole can also be defined

EE 322C
Data Structures

Lecture 6

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

Announcements

- Assignment 1 due next Monday. Use the drop box on the black board for our class
- **REMEMBER**: Exam 1 date – Sep 27
- Topics of the day
  - More on Classes and objects
    - Contruction/destruction
  - Inheritance

Classes and Objects

- A *Class* is a programmer defined type that provides ***modularization***, e***ncapsulation*** and ***abstraction***. This allows data and its operations to be accessed only through the defined interface.
- A Class
  - Is a modularization structure
  - Localizes related data and its operations together
  - Provides an abstract interface
    - Separates the interface and implementation
    - Logically hides the implementation details from the user – ie, provides a black box

Classes and Objects

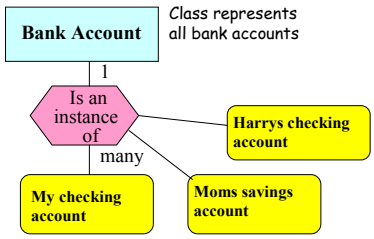
- An instance of a *class* is called an *object*. Objects are *instantiated* by declaring a variable of a given *class*
- Each object is independent of other objects even though they are instantiated from the same class (just as for built in data types)
- Member functions are called by using
  - the dot operator with the object name
  - or in case of an object pointer, the "->" is used.

Modeling Bank Accounts (OO)

What are the common attributes of and behaviors associated with all bank accounts?

Attributes:
- Account #
- Owners name
- Balance

Behaviors
- Deposit $
- Withdraw $
- Check current balance



An object is a specific member of the class

Abstract Data Type

An abstract data type (ADT) is a high level description of a new data type to be implemented. The kind of objects and their common attributes and operations are described, in general. E.g. This is a definition of an abstract class of related bank account objects.

Bank Account: a bank account is a place where someone keeps their money

Common Attributes:

balance - the current value in $$ in this account

owner - the name of the person that this account belongs to

account_ID - a unique number assigned by the bank that identifies this account

Common Operations:

deposit - updates the current balance by adding in a given amount
post condition: the new balance is increased by the amount of $$

withdraw - updates the current balance by subtracting the given amount
post condition: the new balance is decreased by the given amount of $$
pre condition: the current balance must have at least the given amount

get_balance - returns the value of the current balance of the account

Class Definition - version 1

```
class BankAccount // the template for bank account objects
{
  private:  // here are the instance variables
       int     accountNumber;
       string  ownersName;
       double  balance;
  public:
    // all the member  functions are defined here
    BankAccount(int account, string name,double initialBalance )
      {
        accountNumber = account;
        ownerName = name;
        balance = initialBalance;
      }
    ~BankAccount ( ) { } // do nothing destructor
    void BankAccount:: deposit(double amount)
         {  balance = balance + amount; }
    void BankAccount:: withdraw(double amount)
         {  balance = balance - amount; }
    double BankAccount:: getBalance()  {  return balance;  }
}

// can also declare global object variables here - e.g. ob1,ob2;
```


Class Definition - version 2

```
class BankAccount // the template for bank account objects
{
  private:  // or protected if we want them inherited later
       int      accountNumber;
       string   ownersName;
       double   balance;
  public:  // just the function prototypes here
       BankAccount(int, string ,double );
       ~BankAccount( );
       void deposit(double);
       void withdraw(double);
       double getBalance();
}
  // all the member functions are fully defined here
  BankAccount :: BankAccount(int account, string name, double initialBalance )
      {  accountNumber = account;
         ownerName     = name;
         balance       = initialBalance;
      }
  BankAccount ::~BankAccount ( ) { } // do nothing destructor
  void BankAccount:: deposit(double amount)
       {  balance = balance + amount; }
  void BankAccount:: withdraw(double amount)
       {  balance = balance - amount; }
  double BankAccount:: getBalance()  {  return balance;  }
```


Creating and Accessing Objects

- To declare a variable and instantiate an object of that type we say:

```
BankAccount  my_account;                        // or
BankAccount  my_account(10);                    // or
BankAccount  my_account(1, "me", 0 );           // or
BankAccount* my_accountptr = new BankAccount; // or
BankAccount* my_accountptr = & BankAccount;
```

- To apply a member function to an object of that class we say:

```
my_account.deposit (1000);                      // or
my_account.withdraw (100);                      // or
double current = my_account.getBalence ( );     // or
my_accountptr->getBalence ( );
```

- To assign one object to another we say, e.g. :

```
BankAccount your_account = my_account;
```


Access Specifiers

- Member data and functions of a class are divided into access control categories:
  - *private* (scope only within the class definition) - private members are known only inside of objects of that class
  - *public* (scope within it's object's declaration scope), typically in OOD the public parts provide a controlled interface to the private parts
  - *protected* (similar to private, but access is also available to derived classes and objects)
  - In addition *friend* is an access specifier that provides an explicit access by a an external function (or class) to private and protected members of the class in which it is declared to be a friend.

Access Specifiers

- Purpose of access control is not for security, but just for containment/isolation.
  - Keep the client programmer' s hands off those data and private member functions that they are not supposed to touch
  - Also to ease the life of the programmer because he/she doesn' t have to learn too many interfaces
  - Allow the library designer to change the internal structure of a class, without notifying client programmers - E.g., change a variable name

Private Data

- You can't directly access private data from outside the class:

```
harrysChecking.balance = 1000;// ERROR, why?
```

- Must use the public interface for all access:

```
harrysChecking.deposit(1000);
harrysChecking.withdraw(1000);
harrysChecking.getBalance();
```

- Hiding implementation = *abstraction*
  - Makes it easy to change implementation

Scope of Variables

Scope - a definition of where a variable is accessible or visible

Global variable - known everywhere

Local variable - The scope of a local variable starts from its declaration statement and continues to the end of the block that contains it. A local variable must be declared before it can be used.

Instance variable - The scope of an instance variable depends on its access scope specifier.

- ✓ **public** - The variable is visible anywhere
- ✓ **private** - The variable can be accessed only within the declaring (containing) class and objects.
- ✓ **protected** - the variable is private but can be accessed by any derived class and objects

Class Implementation Issues

- The interface is the set of member functions that provides external access to the data members of the class
- Each instance of a class is independent of other instances. However, there are ways to provide sharing. One simple way is to define a data member as ***static***. Variables declared as static in a class are essentially class level variables and have the same value across all objects from that class. Functions can be ***static*** too.
- The member functions can be defined (implementation)either within the class declaration or defined external to it. External implementation of the member functions helps keep the interface (the declaration) independent from the actual implementation, but requires more housekeeping

Class Implementation Issues

- Generally, the class declaration is kept in a header file, which is then *included* in any file that uses that class. Keeping implementation separate avoids any unnecessary recompilations when an implementation of a member function is changed.
- External definition of member functions must be provide the scope resolution, class name followed by "::"
- Classes can also be defined inside of or local to a function

Constructors and Destructor

- Like other variable declarations with built-in types, objects can be initialized at instantiation (creation) time. This is done by calling a constructor function defined in the class.
  - BankAccount my_account (1, "me", 0);
- Every class has an implicit default constructor
  - BankAccount ( );
- Constructor functions can be overloaded and a particular constructor can be invoked at object creation time based on the initialization argument value(s) provided
- The Destructor is called when the object space is deallocated. In addition, the delete function can be used for dynamic deallocation, e.g.

```
BankAccount::~BankAccount()
{   //no arguments and no overloading
    delete [ ] a;          // e.g. deletes the array a
}
```

- The constructors don't have a return type and the destructor has no arguments

Passing and Returning Objects

- Call by value is used by default for passing objects as arguments to a function - a copy is made
- A function may return an object as its result - a temporary object is used for the return value
- You can pass a reference or a pointer to the object if you want the changes (side-effects) to be reflected after returning

Inheritance

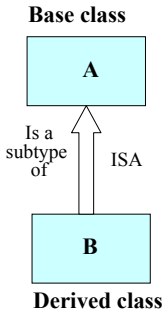
- Inheritance allows one object to acquire the properties of another object (from its ancestor)
- Its purpose is to allow us to build classification schemes,e.g.
  - The term ***ISA*** means is a specific subtype of
  - A red delicious apple **isa** apple **isa** fruit **isa** food
- Makes it possible for an object to be a specific instance of a more general case
- A general class defines common traits, and a more specific subclass only adds those things that are unique to the subclass (or subtype)
- One of the three cornerstones of OOP

Inheritance in C++

- C++ organizes classes into hierarchies.
  - The classes defined by your program
  - The classes in the various libraries too
- Classes inherit instance variables and functions from classes above via the superclass/subclass relationship
- Classes can extend inherited characteristics by
  - adding instance variables and functions and
  - overriding inherited functions.
- Inheritance is an important mechanism for *reusing code*.
- Once a class has been defined it serves as a template for creating objects (instances) of that class - one type of which might be a subclass

Inheritance Among Classes

- A and B are classes that are hiearchically related
- What does the class B **inherit** from A?
  - All the non-private functions and variables
  - Accessability properties
- class B can also define additional functions and variables, and can also override the functions inherited from A

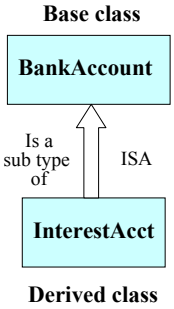


Inheritance Example

- Let's suppose that we need to create a special type of bank account for our program - one which allows interested to be made on the outstanding balance (in general, not all bank accounts are interest bearing).
- Since we already have BankAccount defined, let's use that to define a new subclass

Inheritance Example

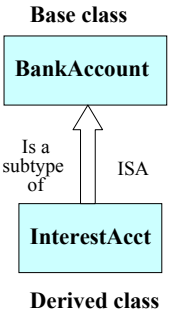
- Let's suppose that we need to create a special type of bank account for our program - one which allows interested to be made on the outstanding balance (in general, not all bank accounts are interest bearing).
- Since we already have BankAccount defined, let's use that to define a new subclass



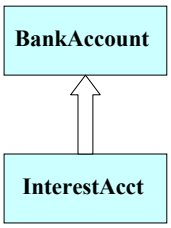
Inheritance Example

- What is **inherited** ?
  All non private functions and variables
- `InterestAcct` can also define additional functions and variables
  E.g. **`addInterest ( )`**
- and can also override the functions inherited
  - A new version of `withdraw ( )`

```
class InterestAcct: public BankAccount
{
  // additional stuff defined in here
}
```

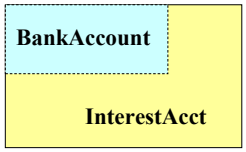


Inheritance – Be Careful



Reuse view of inheritance:
- Separate classes
- Reuses superclass
- Inherits from superclass
- Extends superclass

Inheritance – What Really Happens



Realistic view of inheritance:
- Sort of separate classes
- Incorporates superclass
- Inherits from superclass
- Extends superclass
- Has access to superclass

Inheritance is a two edged sword:
- A very useful mechanism
- But can break certain assumptions
  - No effects on superclass
  - Reuse for free without retesting

EE 322C
Data Structures

Lecture 7

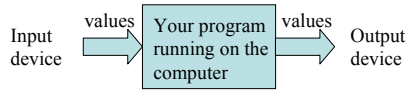
Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

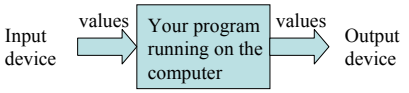
Announcements

- Exam 1 date – 27 September
- Topics of the day
  - Stream I/O
  - File I/O

C++ Input/Output



C++ Input/Output



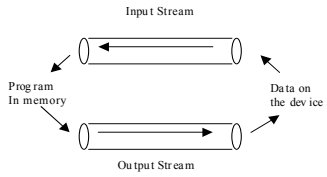
We use the ***iostream*** classes for simplifying the process of inputting and outputting values into/out of the program from/to the std devices.This allows us to use the four predefined streams:

- **cin** - standard input stream - keyboard
- **cout** - standard output stream - screen
- **cerr** - standard error message output - screen
- **clog** - buffered version of cerr - screen

One of the iostream classes (***ios***) contains most of the features that we need use

Streams

- A stream (object) is an abstraction of the continuous one-way flow of data between a program and an external device (e.g., keyboard or screen) or secondary storage device (e.g. disk).
- The stream classes can be categorized into two types: *byte streams* and *character streams*.
- They provide a buffered, continuous sequence of chars or bytes - transferred one at a time



I/O Operators, Functions and Flags

Operators

- **>>** - the extraction operator
  - `cin >>` *variable_name;*
- **<<** - the insertion operator
  - `cout << "hello, world" << '/n';`

Functions

- `width( ), precision( ), fill( )`
- Many manipulator functions

Formatting flags (18 of them)

- Use `setf( )` to set them
- Use `unsetf( )` to clear them

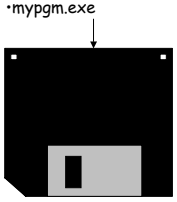
== > See cppreference web page for more information

Files

A **file** is —

1. A named collection of related data stored on a secondary storage device, such as disk, diskette, tape, CD, etc.
   - Permanent — data survives indefinitely.
2. A software object to model the transfer of data between the secondary storage device and main memory.

Files on here: e.g.
- tfgrading.cpp
- ass9data.txt
- myresume.doc
- mypgm.exe



Floppy disk

Preparing to Use External Files

- Use `fstream` library of classes: it will link your files to a stream of the appropriate type
  - `ifstream` - for input
  - `ofstream` - for output
  - `fstream` - for both input/output
- For example:
  - `ifstream my_input_file;`
  - `ofstream my_output_file;`
- Now we need to open it and associate it with a real file name on the device - use the `open( )` function for that
  - `my_output_file.open ("actual file name", ios::out);`
  - `my_input_file.open ("actual file name", ios::in);`
- Shortcut form that declares and opens file is
  - `ifstream my_input_file ("actual file name");`
- We have to close the file when done processing it
  - `my_input_file.close ( );`

Reading and Writing Files - Example 1

```
// Purpose: copy one file to another, a line at a time
#include <string>
#include <iostream>
#include <fstream>  // includes the file I/O library
using namespace std;
int main()
{   ifstream infile  ("Scopy.txt"); // Open for reading
    ofstream outfile ("Scopy2.txt");
       // Open for writing string s;
    while(fstream::getline(infile, s))
       // Discards newline char
       outfile << s << "\n"; // ... must add it back
    return 0;
} // all files get closed at the end of program
```


What if the file can't be opened?

```
if (! infile)   // false if not open
{
   cerr << "open failed" << endl;
   // etc.
}
// or
if (! infile.is_open ( ))  // true if open
{
   cerr << "not open" << endl;
   // etc.
}
//  or
if ( infile.eof ( ) ) // detects if end of file reached
{
   // do whatever
}
```


File Processing Functions

- Can still use >> and <<
  - Be careful with >> as character translations can occur on input
- `get( )` and `put( )` - char at a time
- `read( )` and `write( )` - binary blocks
- `getline( )` - strings
- `flush( )` - flushes the buffer

Coordinating Input and Output

- Plan the format of your output file so that the data can be read back in easily.
- For example, put a space between numbers, so an input process can read them easily

Product Inventory Creation

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{ ofstream out("INVNTRY"); // output, normal file
  if(!out)
  {  cout << "Cannot open INVENTORY file.\n";
      return 1;
  }
  out << "Radios " << 39.95 << endl;
  out << "Toasters " << 19.95 << endl;
  out << "Mixers " << 24.80 << endl;
  out.close();
  return 0;
}
```


Inventory Example Readback

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main()
{ ifstream in ("INVNTRY"); // input
  if(!in)
  {  cout << "Cannot open INVENTORY file.\n";
     return 1;
  }
  string item;
  float cost;
  do
  {  in >> item >>  cost;
     cout << item << " " << cost << "\n";
  } while (!in.eof ( ));
  in.close();
  return 0;
}
```


File IO Brain Teaser

```
#include <iostream>
#include <fstream>
#include <string>
using  namespace std
int main( )
 {   ifstream  infile ("C:/ee322c/IOtest/test1.txt");
     ofstream  outfile ("C:/ee322c/IOtest/output1.txt");
     string instring;
     getline (infile, instring);
     cout << " The line in the file is: " << "\n" << instring;
     int i;
     infile >> i;
     cout  <<  "The integer in the file is " <<  i << "\n";
     double d;
     infile >> d;
     cout << "The double in the file is " << d << "\n";
     outfile << instring;
     outfile << d << "\n";
     outfile << i << "\n";
     outfile.close();
     infile.close();
     return 0;
 }  // end of main
```


Sample Files for Example

Here are contents of test1.txt used as input in the sample program

My name is Colin.
197608
3.1415926

(Note: There are no empty lines between text or number lines.)

The contents of output1.txt after the sample program is executed will be:

Sample Files for Example

Here are contents of test1.txt used as input in the sample program

My name is Colin.
197608
3.1415926

(Note: There are no empty lines between text or number lines.)

The contents of output1.txt after the sample program is executed will be:

My name is Colin.3.1415926
197608

Exam Rules and Expectations

- Exam will start at exactly 5PM and finish at exactly 6:15PM on Monday
- Bring one or more pencils
- No other devices are allowed
- No questions answered during the exam

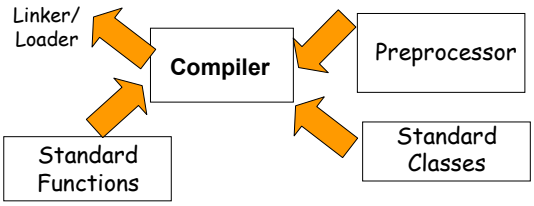
```
if (confused) then
{
  make an assumption
  write it down
  proceed to answer the question on that basis
}
```

- Prepare - eg, use cppreference.com and the other links provided on the web site.

Topic Areas

- Software engineering principles
  - Design techniques (OO vs functional), ADTs, algorithms
- C++ language
  - Compiler, builtin features, function library, class library, preprocessor
- Simple data structures
  - string, etc.
- Defining new and richer data types
  - Safe array, bank accounts, etc

The Parts of C++



C based I/O
String and char manipulations
Math functions
Time,date,location functions
Dynamic allocation
Utility functions
Wide character functions

C++ I/O
String
STL classes
Numerics
Exception handling
Misc.

Topics Covered (1)

- What is SWE, programming, SWE tracks in ECE, etc.
- How to think about and solve programming problems
- Modularity, encapsulation and abstraction
- OO thinking
- C and C++ basics (EE 312 topics assummed)
- Strings
- Building your own data types
- Booleans
- Functions
- Overloaded functions
- Structs, Bit fields, Unions. Enumerations, Typedef

Topics Covered (2)

- Classes and objects
- Construction/destruction of objects
- Subclasses and Inheritance
- Stream and File I/O
- Arrays of objects
- Pointers
- References
- Dynamic memory allocation
- Polymorphism & Overloaded operators
- Generic functions and classes
- Exception handling

Other Hints and Tips

- Make sure you understand the examples used in class
- Make sure you understand your assignment solution

EE 322C
Data Structures

Lecture 8

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm in ACES 5.1.4

Announcements

- Will **NOT** have exam next Monday
  - I will be out next week
  - Matt will give lectures on MW
  - Will be rescheduled so will get feedback before Q-Drop date (October 22)
- Topics of the day
  - **Arrays of objects**
  - **Pointers**
    - Concepts
    - Basic operations
    - Advanced operations
    - Dangers and problems
  - **References**
  - **Dynamic memory allocation**

An Inventory of Products

Envision a retail store with lots of different products on the shelves that we need to keep track of

Product

We can model "product" as a general class of all product objects
- We will design it to be "object oriented"

An Inventory of Products

Envision a retail store with lots of different products on the shelves that we need to keep track of

Common operations:
- change price
- Change score

Product

Common attributes:
- name
- price
- score

We can model "product" as a general class of all product objects
- We will design it to be "object oriented"

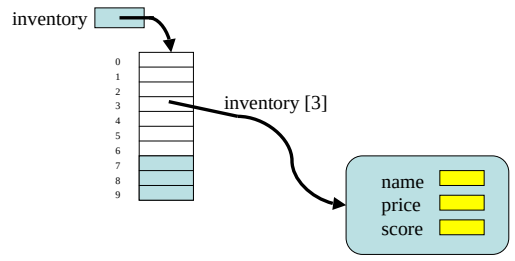
Product Class

```
class Product // models each  product in a store
  {private:
     string name;
     double price;
     int score;
     static int numberProducts = 0;
        //keeps track of the # of products
   public:
     Product(string theName, double thePrice,
             int theScore)
     { name  = theName;
       price = thePrice;
       score = theScore;
       numberProducts++;
     }
     ~Product( ) {numberProducts--;}  // destructor
     // more functions on the next slide
```


Product Class

```
 // instance variable accessor functions
     int getScore( )     { return score; }
     double getPrice ( ) { return price;}
     string getName ( )  { return name; }

 // instance variable change functions
      void changeScore (int newScore )
           { score = newScore; }
      void  changePrice (double newPrice )
           { price = newPrice;}
};
```

Array of Product Objects



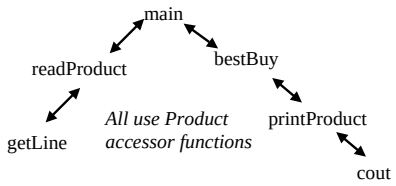
Best Product Example

- Fill up the product inventory
- Then search the inventory and find the "best buy(s)"
- A "best buy" is indicated by the largest score to price ratio
- What would be a good modular design for our solution?

Best Product Example

- Fill up the product inventory
- Then search the inventory and find the "best buy(s)"
- A "best buy" is indicated by the largest score to price ratio

Function Calling Hierarchy



Pointer Concepts

- The pointer type supports indirect addressing
- A pointer holds a memory address rather than a value - what is at the memory address is the value of interest
- Pointer variables have to be declared along with the type of data that they point to. E.g.
  - `int *p; // means that pointer p points to an int`
  - Never use `int* p;` - its bad style
- Mixed type pointer manipulations are not allowed
  - `int *p;`
  - `double *q;`
  - `p = q; // not allowed`
- Basic unary operations are: & (the address of) and * (the contents at an address)

Your Computer's RAM

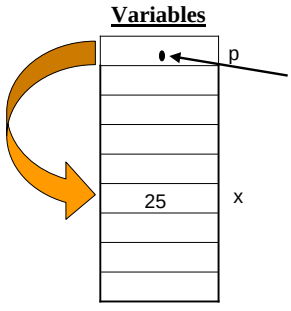
Memory is organized into a sequence of "bytes" - each byte has it's own memory ***address*** and contents

Remember that variable names are actually symbolic memory addresses

RAM = random access memory

Address	Contents
0)	01011100
1)	01011100
2)	01011100
3)	01011100
.	
.	
.	
.	00000001
	00101111

Pointer Implementation



A pointer variable contains the ***memory address*** of another variable as its value

```
int *p = 0; //null
int x:
p = &x;
x = 25;
```

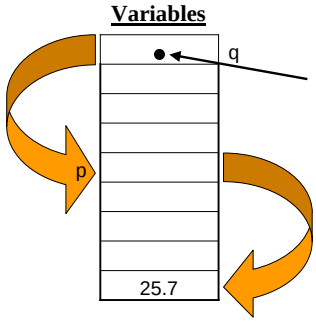

RAM = random access memory

Pointer Manipulations

- Basic operations are: & (the address of) and * (the contents at that address)
- Pointer arithmetic is done relative to the base type, e.g.:

```
int *p, *q;  // int *p =0, *q =0; would be better
int x[4] = {25, 37, 77, 99};
p = &x[0];   // or just p = x when x is an array
q = p;
q++;
if (p < q)
  cout << "p points to a lower address" <<endl;
cout << *(p+2)  // faster but uglier
```


Multiple Indirection



A pointer variable contains the ***memory address*** of another pointer variable as its value

```
double **q;
double *p;
double x = 25.7
p = &x;
q = &p;
cout << **q << endl;
```


RAM = random access memory

Object Pointers

```
class ExampleClass
{ private: int i;
  public:
   ExampleClass(int j) { i=j; }
   int get_i() { return i; }
};
int main()
{
  ExampleClass *p, objects[4] = {1,2,3,4};
  p = &objects; // get address of objects
  cout << p->get_i() << endl; // use -> to call
  get_i()
  p = p +2;
  cout << p->get_i() << endl;  // which object?
  return 0;
}
```


Object Pointers

```
class ExampleClass
{ private: int i;
  public:
   ExampleClass(int j) { i=j; }
   int get_i() { return i; }
};
int main()
{
  ExampleClass *p, objects[4] = {1,2,3,4};
  p = &objects; // get address of objects
  cout << p->get_i() << endl; // use -> to call get_i()
  p = p +2;
  cout << p->get_i() << endl;  // which object?
  return 0;
}
```

Also - You can also use a base class pointer to point to a derived class object, but not vice versa - be careful with pointer arithmetic because the compiler thinks its pointing to a base class object

The `this` pointer

- The object which invokes a member function is an implicit argument passed as a pointer which is called by the keyword `this`
- ***E.g.*** if I call a member function
  - `myaccount.withdraw (200);`
  - Inside of the withdraw function I can refer to the calling object as `this`

```
double withdraw (double amount)
{ cout  << this->balance  << endl;
…
}
```

- Doesn't work for static or friend functions

Problems with Pointers

- Beware the wild pointer
  - It's the most difficult bug to find
- A bad pointer can get you garbage or cause you to write over code or even the O/S
- Common errors
  - Unitialized pointers
  - `int x =10, *p;`
  - `*p = x;`
  - Misunderstood usage of pointers
  - `int x =10, *p;`
  - `p = x;  // oops, p = &x`

Problems (cont)

- Incorrect assumptions about where variables are located

```
int x, y;
int *p, *q;
p = &x;
q = &y;
if (p < q) …
```

OR

```
int x[10], y[10];
int *p, t;
p = x;
for (t=0; t<20; t++)
  *p++ = t;
if (p < q) …
```


Problems (cont)

```
/*
   This program has a logic bug - can you find it?
   It's purpose is to output the hex equivalent of a
   sequence of chars
*/
#include <string>
#include <iostream>
#include <cstdio>
int main( )
{
   char *p1=0; char s[80];
   p1 = s;
   cout.setf(ios::hex);
   do
   {
        gets(s); // read a string of chars, puts a null at end
                 // output the hex equivalent of each character
       while(! *p1=="") {cout  << *p1 << endl;  p1++;}
   } while(!strcmp(s, "QUIT")); // sentinel check
  return 0;
}
```


References

- A special kind of pointer that is used for passing arguments to and returning values from functions
  - There is also an independent reference capability but it is not useful and we won't learn about it
- Call by reference is faster than call by value
- Restrictions are:
  - You cannot obtain the address of a reference
  - You cannot create an array of references
  - You cannot reference a bit field
  - Null references are prohibited

Reference Parameters

```
/* example of using reference parameters */
#include <iostream>
using namespace std;
void swap(int &i, int &j);
int main( )
{
   int a =1, b =2;
   cout << "a and b: " << a << " " << b << endl;
   swap(a, b); // no & operator needed
   cout << "a and b: " << a << " " << b << endl;
   return 0;
}
void swap(int &i, int &j)
{
   int t;
   t = i;
   i = j;
   j = t;
}
```

What's the output of this program?

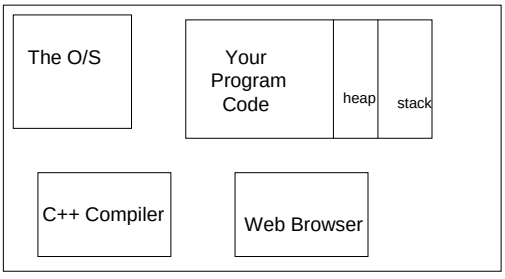
Returning a Reference

```
#include <iostream>
using namespace std;
char &replace(int i); // returns a reference
char s[80] = "Hello There";
int main( )
{
  replace(5) = 'X';
  cout << s << endl;
  return 0;
}
char &replace(int i)
{ return s[i]; }
```

What's the output of this program?

General Memory Mgt. Scheme

- The responsibility of the O/S



Dynamic Memory Allocation

- Global variables are allocated space at compile time. Local variables are allocated space on the stack when the block is entered. Dynamic allocations occur during run time when a statement is executed and use the heap.
- Pointers are needed to point to the space dynamically allocated
- Useful in applications where space is constrained and must be managed by the programmer
- Operators are:
  - new - allocate memory NOW and return a pointer to it
    - *pointer_variable* = new *type;*
  - delete - deallocate (free) memory NOW
    - delete *pointer_variable;*
  - malloc and free are obsolete

Variable Example

```
#include <iostream>
#include <new>
using namespace std;
int main( )
{
   int *p;
   p  = new int;  // allocate space for an int
   *p = 100;
   cout << "At " << p << " ";
   cout << "is the value " << *p << endl;
   delete p; // free the space for an int
   return 0;
}
```


Array Example

```
#include <iostream>
#include <new>
using namespace std;
int main( )
{
   int *p, i;
   p = new int [10]; // allocate 10 integer array
   for(i=0; i<10; i++) p[i] = i;
   for(i=0; i<10; i++) cout << p[i] << " ";
   delete [ ] p; // release the array
   return 0;
}
```


Objects Example

```
. . .
int main( )
{
  BankAccount  *p;
  p = new BankAccount
      (12387, "Ralph Wilson", 2000.00);
  double s = p->getBalance( );
  cout << "The balance is: " << s << endl;
  delete p;
  return 0;
}
```

EE 322C
Data Structures

Lecture 9

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

Announcements

- Class Email List
  - **READ YOUR ECE EMAIL**
- Topics of the day
  - Inheritance nuances
  - Dynamic memory allocation
  - Random numbers
  - Polymorphism

Access Control

Access to base class members from a derived class:

Base Class Members

Derived Class Access	Private	Protected	Public
Private	no access	access as private	access as private
Protected	no access	protected access as before	access as protected
Public	no access	protected access as before	public access as before

Constructors, Destructors Nuances

- Constructors and destructors are not inherited
- Constructors are called in the order of hierarchy starting from the base to the derived classes
- Destructors are called in the reverse order automatically
- Arguments can be passed through the initialization lists from a derived class constructor to a base class constructor, e.g.

```
//constructor in the derived class
derived (int i, int j):base (i) { // stuff  };
// base(i) is base class constructor
```

- No need to specify the base class in the constructor if it doesn't have to be called explicitly since the default constructor is called automatically

Dynamic Memory Allocation

- Global variables are allocated space at compile time. Local variables are allocated space on the stack when the block is entered. Dynamic allocations occur during run time when a statement is executed and use the heap.
- Pointers are needed to point to the space dynamically allocated
- Useful in applications where space is constrained and must be managed by the programmer
- Operators are:
  - `new` - allocate memory NOW and return a pointer to it
    - *pointer_variable* = `new` *type;*
  - `delete` - deallocate (free) memory NOW
    - `delete` *pointer_variable;*
  - `malloc` and `free` are obsolete

Variable Example

```
#include <iostream>
#include <new>
using namespace std;
int main( )
{  int *p;
   p = new int;  // allocate space for an int
   *p = 100;
   cout << "At " << p << " ";
   cout << "is the value " << *p << endl;
   delete p; // free the space for an int
  return 0;
}
```

Array Example

```
#include <iostream>
#include <new>
using namespace std;
int main( )
{  int *p, i;
   p = new int [10]; // allocate 10 integer array
   for(i=0; i<10; i++ )  p[i] = i;
   for(i=0; i<10; i++) cout << p[i] << " ";
   delete [ ] p; // release the array
   return 0;
}
```


Objects Example

```
. . .
int main( )
{ BankAccount  *p;
  p = new BankAccount
      (12387, "Ralph Wilson", 2000.00);
  double s = p->getBalance( );
  cout << "The balance is: " << s << endl;
  delete p;
return 0;
}
```


Simulating Randomness

- What is a pseudo-random number?
- What's a sequence of these?
- Library functions are in cstdlib
  - `#include <cstdlib>`
- `srand ( )` - seeds `rand ( )`, sets the starting point
- `rand ( )` - returns a random integer between 0 and `RAND_MAX`
- `RAND_MAX` is the largest value that can be returned by `rand ( )` - it will be $(2^{31} - 1)$

Simple Examples

- Flip a coin

```
while (notDone)
{ double x = rand ( );
   if ( x / RAND_MAX <= .5)
        cout << "heads";
   else cout << "tails";
. . .
}
```


Random Numbers

```
/* Random (magic) number guessing program */
#include <iostream>
#include <cstdlib>
using namespace std;
int main(void)
{ int magic, guess;
  bool correct = false;
  unsigned const int seed = rand( );
  srand (seed);
  cout << "The number is between 0 and " << RAND_MAX << endl;
  magic = rand( );  // generate the random magic number
  while (!correct)
  {  cout << "What is your guess at the magic number: " << endl;
     cin >> guess;
     if(guess == magic)
     {  cout << "** Right ** " << magic << ;" is the magic number"
              <<endl;
        correct = true;
     }
     else if(guess > magic)
       cout << "Wrong, too high" << endl;
     else  cout << "Wrong, too low" << endl;
  }  // end while
  return 0;
}
```


Intro to Polymorphism

- ***Polymorphism***, which is another key concept in OO programming, is the ability to control access to a general class of actions through one interface. The specific action selected is determined by the contextual situation
  - E.g. the thermostat in your house doesn't care what type of furnace you have
- There is both compile time and run time polymorphism in C++
  - At compile time - overloaded functions and operators
  - At run time - inheritance and virtual functions
  - The C++ language also uses this extensively in its library system
- The design goal is to define a class hierarchy that moves from the most general to specific (I.e. base class to derived classes). We use Polymorphism, Inheritance and Encapsulation to do this

Virtual Functions

- Is declared within a base class and redefined (or overridden) by a derived class
  - The redefinition creates a new specific function
- Identified by the keyword `virtual`
  - `virtual` *return-type function-name* (*params*) {*body*}
- When a base pointer points to a derived object that contains a virtual function then C++ determines at run time which version of the function to call based on the type of object pointed to (polymorphism)
- A pure virtual function has no definition in the base class and therefore must be defined by the derived classes
- The virtual attribute can be inherited
- Virtual functions are hierarchical

Example

```
#include <iostream>
using namespace std;
class base
{ public:
  virtual void vfunc( )
   {cout << "This is base's vfunc." <<endl;}
};
class derived1 : public base
{ public:
  void vfunc( )
   { cout << "This is derived1's vfunc." << endl; }
};
class derived2 : public base
{ public:
  void vfunc( )
   {cout << "This is derived2's vfunc." <<endl;}
};
```


Example cont.

```
/* program that demonstrates the use of
  virtual functions */
int main( )
{ base b, *p;
  derived1 d1;
  derived2 d2;
  p = &b;       // point to base class
  p->vfunc( );  // access base's vfunc()
  p = &d1;      // point to derived1
  p->vfunc( );  // access derived1's vfunc()
  p = &d2;      // point to derived2
  p->vfunc( );  // access derived2's vfunc()
  return 0;
}
```


Pure Virtual Function

```
#include <iostream>
using namespace std;
class number
{  protected: int  val;
   public:
     void setval(int i) { val = i; }
     virtual void show( )  = 0;
        // show() is a pure virtual function
};
class hextype : public number
{ public: void show( ) {cout << hex << val << endl;}
};
class dectype : public number
{ public: void show( ) {cout << val << endl;}
};
class octtype : public number
{public: void show( ) {cout << oct << val << endl;}
};
```

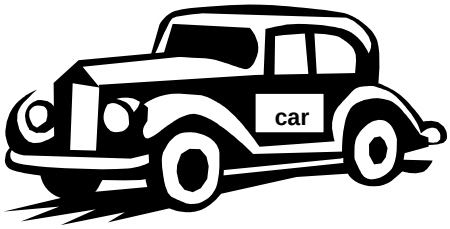

Pure Virtual Function - cont.

```
/* demonstrate the use of a pure virtual function */
int main( )
{ dectype d;
  hextype h;
  octtype o;
  d.setval (20);
  d.show (  );   // displays 20 - decimal
  h.setval (20);
  h.show (  );   // displays 14 - hexadecimal
  o.setval (20);
  o.show (  );   // displays 24 - octal
  return 0;
}
```


Abstract Classes

- Classes that contain one or more pure virtual functions are called abstract because no objects can be instantiated
- They serve the role of an incomplete type that is used as a foundation for derived classes which will be complete

Generics



Generic Functions

- Defines a general set of operations that will be applied to various types of data
- Created using the keyword `template`
- Remember this example from last time that swaps the values of a and b

```
// in main
  int a =1, b =2;
  swap(a, b);
 . . .
void swap(int &i, int &j)
{ int t;
  t = i;
  i = j;
  j = t;
}
```


Generic Functions

- Defines a general set of operations that will be applied to various types of data
- Created using the keyword `template`
- Remember this example from last time that swaps the values of a and b

```
// in main
  int a =1, b =2;
  swap(a, b);
 . . .
void swap(int &i, int &j)
{ int t;
  t = i;
  i = j;
  j = t;
}
```

It only works for ints - suppose we also want it to work for doubles, chars, etc.
How can we turn it into a generic function?

Generic Swap

```
/* the purpose of this program is to demonstrate how to swap values
of 2 variables of any data type using a generic function to do so
*/
#include <iostream>
using namespace std;
// This is a generic function template.  X is a generic data type.
template <class X> void swapargs(X &a, X &b)
{ X temp;
  temp = a;
  a = b;
  b = temp;
}
int main()
{ int i=10, j=20;
  double x=10.1, y=23.3;
  char a='x', b='z';
  swapargs(i, j); // swap integers
  swapargs(x, y); // swap floats
  swapargs(a, b); // swap chars
  return 0;
}
```


More on Generic Functions

- They can be overloaded
  - They perform the same actions - only the data type can differ
- You can mix standard parameters with generic parameters in the function template

Generic Classes

- Used when a class contains logic that can be generalized across different data types
- Defines all the functions used by that class
- the actual type of data being manipulated will be passed as a parameter when an object of that class is created

EE 322C
Data Structures

Lecture 10

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

Announcements

- Topics of the day
  - Overloaded operators
  - Generic classes
  - Exception handling

Overloaded Operators

- You can overload operators as well as functions - this expands the types of data that it can be applied to
  - For example, that is how string concatonation is defined
- You can use overloaded operators in expressions just like you use the built-in ones
- How it works is defined inside of the class of objects it will be used with
- Syntax is:

  return-type [*class-name*::] `operator` X (*arg-list*)
  `{ // implementation body }`

Class with an Overloaded "+"

```
class location
/* this class represents location points on an 2D x-y grid  */
{ private: int xCoordinate, yCoordinate;
  public:
    location() {}
    location(int x, int y)
    { xCoordinate = x;
      yCoordinate = y;
    }
  void show()
   {   cout << xCoordinate << " ";
       cout << yCoordinate << "\n";
   }
   // define overloaded  "+" operator for locations.
  location operator+(location op2)
   {  location temp;
      temp.xCoordinate = op2.xCoordinate + this.xCoordinate;
      temp.yCoordinate = op2.yCoordinate + this.yCoordinate;
      return temp;
   }
};  // end of class location definition
```


Using the Overloaded "+"

```
#include <iostream>
using namespace std;
// class location definition goes in here
// main program that uses location
int main()
{ location point1(10, 20), point2( 5, 30), point3(0,0);
  point1.show(); // displays 10 20
  point2.show(); // displays 5 30
  point3 = point1 + point2;
       // use the overloaded + operator
  point3.show(); // displays 15 50
  return 0;
}
```


Nuances and Restrictions

- The object on the left hand side of a binary operator is the calling object (I.e. this)
- You cannot alter the precedence of an operator
- You cannot change the number of operands that an operator takes
- Operator functions cannot have default arguments
- You cannot overload `. :: .* ?`
- Operator functions can be inherited by any derived class, but it can also be overloaded by the derived class (except for the = operator)
- You can overload new and delete if you want to implement a special dynamic allocation scheme
- Array subscripting, function calling, and class member access are defined as operators `[ ], ( ), ->` They can therefore be overloaded too
- You can overload the comma operator `,`

Generic Functions - again

```
/* the purpose of this program is to demonstrate how to swap values
of 2 variables of any data type using a generic function to do so
*/
#include <iostream>
using namespace std;
// This is a generic function template.  X is a generic data type.
template <class X> void swapargs(X &a, X &b)
{ X temp;
  temp = a;
  a = b;
  b = temp;
}
int main()
{ int i=10, j=20;
  double x=10.1, y=23.3;
  char a='x', b='z';
  swapargs(i, j); // swap integers
  swapargs(x, y); // swap floats
  swapargs(a, b); // swap chars
  return 0;
}
```


Generic Classes

- Used when a class contains logic that can be generalized across different data types
- Defines all the functions used by that class
- the actual type of data being manipulated will be passed as a parameter when an object of that class is created
- A generic (aka template) class declaration looks like

  `template <class` **`Ttype`**`> class` *`class-name`*

  `{ // class definition in here  }`

  `Ttype` is a placeholder for a type name which is specified when the class is instaniated
- An instance gets created in your program when you say

  `class-name` `<`**`actual-type`**`>` *`object-name`*`;`

Generic Class Example

```
// this is a generic (template) class that provides for accessing
//   an array in a safe manner
template <class AType, int size> class arrayType
{ private: AType a[size];
  public:
  arrayType( )
   { register int i;
     for(i=0; i<size; i++) a[i] = i;
  }
// Provide range checking for arrayType by redefining [ ].
AType  &operator [ ] (int i)
{ if(i < 0 || i > size-1)
    {cout << "\nIndex value of ";
     cout << i << " is out-of-bounds.\n";
     exit(1); // terminate the program with a return code of 1
    }
   return a[i];
  }
}; // end of class arrayType
```


Example (cont.)

```
#include <iostream>
#include <cstdlib>
using namespace std;
// the arrayType class definition goes in here
int main ( )
{ arrayType<int, 10> intob;       // create an integer array
  arrayType<double, 15> doubleob; // create a double array
  int i;
  cout << "Integer array: ";
  for(i=0; i<10; i++) intob[i] = i;
  for(i=0; i<10; i++) cout << intob[i] << "  ";
  cout << endl;
  cout << "Double array: ";
  for(i=0; i<15; i++) doubleob[i] = (double) i/3;
  for(i=0; i<15; i++) cout << doubleob[i] << "  ";
  cout << endl;
  intob[12] = 100; // generates runtime error
  return 0;
}
```


Rules and Restrictions for Generic Classes

- Non type parameters can only be int, pointer or reference - which have constants as arguments - because this must be known at compile time
- You can put in a default type in the template header
  - `template <class AType = int> class arrayType { }`
- You can create explicit specializations for a specific data type
- You can use the keyword typename instead of class in the header (I prefer this way)
  - `template <t`**`ypename`** `AType = int> class arrayType { }`
- The Standard Template Library (STL) is built out of template (or generic classes) that have been designed as abstractions for maximum reuse

Why Exception Handling

- We want to manage run time errors in an orderly fashion so as to create fault tolerant software systems
  1. If an exception is not caught, the program terminates.
  2. Advanced error handling can be done that allows for resumption of execution after correcting for an error
  3. Capability to return error information is not limited to just the return type of an operation.
- We want to separate Error Handling Code from Regular Code
  1. Frequent and repetitive testing of return values, and propagation of error return values, is not required.
  2. Using exceptions may be more efficient because the normal execution path does not need to test for error conditions.
  3. Grouping Error Types and Error Differentiation

Why Exception Handling

- We want to propagate Errors Up the Call Stack
  1. The point at which an error occurs is rarely a suitable place to handle it, particularly in library code, but by the time an error code has been propagated to a place where it can be handled, too much contextual information has been lost.
  2. Exceptions bridge this gap - they allow specific error information to be carried from the point where its available to a point where it can be utilized.

Exception Handling - Mechanics

- Uses three keywords: `try`, `catch` and `throw`, e.g.

```
try
 { // block of code to be monitored - could be all of
main ( )
     if (flag)  throw exception-name;
 }
 catch (type1 parameter)
 {  // code to be executed in case of exception
 }
 catch (type2  parameter)
 {  // code to be executed in case of other exception
 }
...
```


Exception Handling - Mechanics

- The catch blocks must be right after the try block.
- The catch executed will be based on the first data type match between the exception name and the catch parameter
- A `catch (…)` will catch any type of a throw and is used like a default catch.
- The whole construct behaves much like a switch block with breaks. After a catch block is executed the next statement after all the try/catches is done. Control does not go back to the try block.
- A throw from a function called within the try block can also cause a catch

Exception Handling Example

```
// this example illustrates prevention of a divide by zero error
#include <iostream>
using namespace std;
void divide(double a, double b);
int main()
{ double i, j;
  do
  { cout << "Enter numerator (0 to stop): ";
    cin >> i;
    cout << "Enter denominator: ";
    cin >> j;
    divide(i, j);  //gracefully handle divide by zero
  } while(i != 0);
  return 0;
}
void divide(double a, double b)
{ try
   {  if(b = = 0.0) throw b; // check for divide-by-zero
      cout << "Result: " << a/b << endl;
   }
  catch (double b)
  {  cout << "Can't divide by zero." << endl;
  }
}
```


Passing an Exception

- Passing an exception by throw is similar to passing an argument to a function, except:
  - On a throw of an exception, the control flow passes over to the appropriate catch block.
  - A copy of the argument is made whether the exception is passed as a reference or a value.
  - The thrown object reference is thrown as a const reference, but it can be caught by a non-const reference; something not allowed in argument passing for functions
  - Implicit type conversion is not done except for `const void*` which will catch all pointer type exceptions and allows inheritance base type conversions

Passing an Exception

- Copy of the object is made because once the exception is thrown, the control passes over from the point of exception and any local object will go out of scope.
  - The copy is made even if the object is declared as static.
  - Any changes to the passed object will therefore only affect the copy

Standard Exceptions

- In `<exception> class`
  - `bad_alloc` - from operator new
  - `bad_exception` - any unexpected exception
  - `bad_cast` - from dynamic cast
  - `bad_typeid` - from dynamic cast
- In `<stdexcept> class` - in function library or run time system
  - run-time errors
    - `overflow_error` - arithmetic overflow occurred
    - `range_error` - an internal range error occurred
    - `underflow_error` - an underflow occurred
  - logic errors
    - `domain_error` - domain error occurred
    - `invalid_argument` - from a function call
    - `length_error` - object created was too large
    - `out_of_range` - function arg not in required range

Other Nuances

- Exceptions thrown with no catch cause termination of program
- Try blocks inside of functions cause reset of handling code
- An exception can be of any type including a class of similar exceptions that you define. This class will be used to create an exception object that describes the error. This object will be used by the exception handling code.
- When catching exceptions from both a base and derived class, catch the derived class first in the catch sequence

Other Nuances

- You can restrict the type of exceptions that a function can throw outside of itself - back to its caller
  - `return-type function-name (parm-list) throw (type-list) { }`
- You can rethrow an exception by saying `throw;` in which case it will look for the next matching catch
- `uncaught_exception ( )` is true until an exception is caught
- You can even override the standard exception handling functions:
  - `unexpected ( )` - which calls `terminate ( )`
  - `terminate ( )` - which calls `abort ( )`
  - `abort ( )` - which causes your program to be flushed out

EE 322C
Data Structures

Lecture 11

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm in ACES 5.104

With Apologies to PEANUTS

Announcements

- Exam I – Next Monday, 11 October 5-6:15
- Office Hours – ACES 5.104
  - Perry: MW 4-5
    - Design issues
  - Hawthorne: MW 4-5, 6:30-7:30
    - Compiler and Environment issues
    - Design issues
- USE YOUR ECE EMAIL ACCOUNTS for class email
- Today:
  - Exceptions
  - Review SE issues

Why Exceptions

- Useful encapsulation and abstraction mechanism
- Separate normal from abnormal processing
  - Normal processing often quite straightforward
  - Abnormal often quite tangled
    - Causes nesting
    - Obscures normal processing
- Necessary for reliable, fault tolerant, robust software systems
  - Often 75%+ code dedicated to reliability
  - 20% of interface errors are errors in handling exceptions

Exceptions - Logic

- Logic of handling exceptions
  - Preclude
    - Guarantee they do not happen
  - Report
    - Up to someone else to do something
  - Retry
    - Works when fault is transient
  - Repair
    - Fix the problem or compensate for the problem
  - Ignore
    - Results are satisfactory even with the exception

Examples

- Nested tangle

```
if (condition1) then
  do something1
  if (condition 2) then
    do something2
    if (condition3) then
      do something3
    else exception3
  else exception2
else exception1
```

- Exceptions

```
Something1
Something2
Something3
Exceptions:
  ex1: exception1
  ex2: exception2
  ex3: exception3
```

Why Exceptions - Details

- We want to manage run time errors in an orderly fashion so as to create fault tolerant software systems
  1. If an exception is not caught, the program terminates.
  2. Advanced error handling can be done that allows for resumption of execution after correcting for an error
  3. Capability to return error information is not limited to just the return type of an operation.
- We want to separate Error Handling Code from Regular Code
  1. Frequent and repetitive testing of return values, and propagation of error return values, is not required.
  2. Using exceptions may be more efficient because the normal execution path does not need to test for error conditions.
  3. Grouping Error Types and Error Differentiation

Why Exception - Details

- We want to propagate Errors Up the Call Stack
  1. The point at which an error occurs is rarely a suitable place to handle it, particularly in library code, but by the time an error code has been propagated to a place where it can be handled, too much contextual information has been lost.
  2. Exceptions bridge this gap - they allow specific error information to be carried from the point where its available to a point where it can be utilized.

Exceptions - Mechanics

- Uses three keywords: `try, catch and throw`, e.g.

```
try
 { // block of code to be monitored - could be all of
main ( )
     if (flag)  throw exception-name;
 }
 catch (type1 parameter)
 {  // code to be executed in case of exception
 }
 catch (type2  parameter)
 {  // code to be executed in case of other exception
 }
...
```


Exceptions- Mechanics

- The catch blocks must be right after the try block.
- The catch executed will be based on the first data type match between the exception name and the catch parameter
- A `catch (…)` will catch any type of a throw and is used like a default catch.
- The whole construct behaves much like a switch block with breaks. After a catch block is executed the next statement after all the try/catches is done. Control does not go back to the try block.
- A throw from a function called within the try block can also cause a catch

Exception Handling Example

```
// this example illustrates prevention of a divide by zero error
#include <iostream>
using namespace std;
void divide(double a, double b);
int main()
{ double i, j;
  do
  { cout << "Enter numerator (0 to stop): ";
    cin >> i;
    cout << "Enter denominator: ";
    cin >> j;
    divide(i, j);  //gracefully handle divide by zero
  } while(i != 0);
  return 0;
}
void divide(double a, double b)
{ try
   {  if(b = = 0.0) throw b; // check for divide-by-zero
      cout << "Result: " << a/b << endl;
   }
  catch (double b)
  {  cout << "Can't divide by zero.” << endl;
  }
}
```


Passing an Exception

- Passing an exception by throw is similar to passing an argument to a function, except:
  - On a throw of an exception, the control flow passes over to the appropriate catch block.
  - A copy of the argument is made whether the exception is passed as a reference or a value.
  - The thrown object reference is thrown as a const reference, but it can be caught by a non-const reference; something not allowed in argument passing for functions
  - Implicit type conversion is not done except for `const void*` which will catch all pointer type exceptions and allows inheritance base type conversions

Passing an Exception

- Copy of the object is made because once the exception is thrown, the control passes over from the point of exception and any local object will go out of scope.
  - The copy is made even if the object is declared as static.
  - Any changes to the passed object will therefore only affect the copy

Standard Exceptions

- In `<exception> class`
  - `bad_alloc` - from operator new
  - `bad_exception` - any unexpected exception
  - `bad_cast` - from dynamic cast
  - `bad_typeid` - from dynamic cast
- In `<stdexcept> class` - in function library or run time system
  - run-time errors
    - `overflow_error` - arithmetic overflow occurred
    - `range_error` - an internal range error occurred
    - `underflow_error` - an underflow occurred
  - logic errors
    - `domain_error` - domain error occurred
    - `invalid_argument` - from a function call
    - `length_error` - object created was too large
    - `out_of_range` - function arg not in required range

Other Nuances

- Exceptions thrown with no catch cause termination of program
- Try blocks inside of functions cause reset of handling code
- An exception can be of any type including a class of similar exceptions that you define. This class will be used to create an exception object that describes the error. This object will be used by the exception handling code.
- When catching exceptions from both a base and derived class, catch the derived class first in the catch sequence

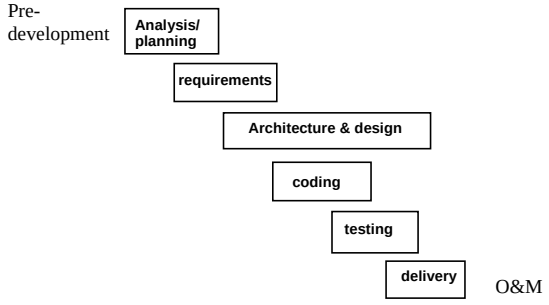
Other Nuances

- You can restrict the type of exceptions that a function can throw outside of itself - back to its caller
  - `return-type function-name (parm-list) throw (type-list) { }`
- You can rethrow an exception by saying `throw;` in which case it will look for the next matching catch
- `uncaught_exception ( )` is true until an exception is caught
- You can even override the standard exception handling functions:
  - `unexpected ( )` - which calls `terminate ( )`
  - `terminate ( )` - which calls `abort ( )`
  - `abort ( )` - which causes your program to be flushed out

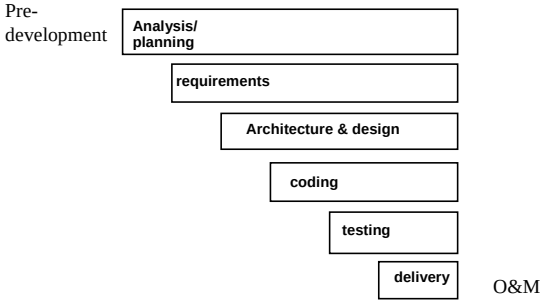
Back to Basic SE

- Typical life-cycle process: waterfall model
- 2 critical domains
  - Problem domain
  - Solution domain
- Our intellectual tools
  - Modularity
  - Encapsulation
  - Abstraction
  - Information hiding
  - Step-wise refinement
  - Virtual machines
- Measurement & Evaluation

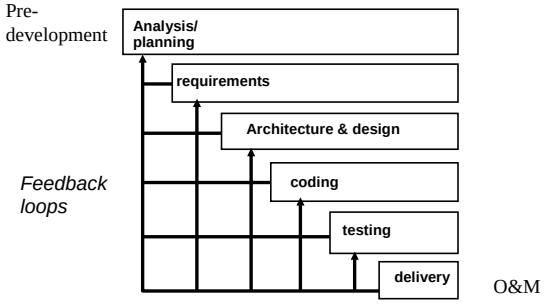
Classic Waterfall



Classic Waterfall



Classic Waterfall



2 Critical Domains

- Problem Domain
  - "The World" – data objects and processing
  - Build a Model
    - Select critical elements from the world for the problem
    - Understand and bound the problem
    - Eg, Scenarios, use cases, etc
  - Define the Requirements
    - What is the system supposed to do
    - Critical implementation constraints
      - Eg, platform, context, languages, etc
  - Critical problem: *understanding the domain*

2 Critical Domains

- Solution Domain
  - "The Machine"
    - modules, data structures and algorithms
  - Find or create critical abstractions
  - Create an Architecture
    - Components
    - Interactions
  - Refine the design
    - Data structures
    - Algorithms
  - Code the representations

Useful Intellectual Tools

- Modularity
- Encapsulation
- Abstraction
- Information hiding
- Stepwise Refinement
- Virtual Machine

Modularity

- Programming is a complex task that is made simpler if you can break the big problem down into smaller pieces
- Modular programming is where big programs get broken down into smaller components called modules - e.g. classes, methods, objects, subroutines, procedures or functions.
- Useful advice/style:
  - Break down a problem into components that each just do one thing and do that thing well
  - (member) Functions and classes provide modularity

Encapsulation

- The logic to do that one thing is "encapsulated" inside of the module.
- Encapsulation localizes
  - Related type, constant, data definitions
  - Related processing – ie, functions, procedures, methods, etc
- Provides a logic to the organization of your program
  - Eg, classes provide both modularity and encapsulation
  - Properly used, functions provide both

Abstraction

- Fundamental in software engineering
- 3 key ingredients
  - Conceptualization
    - finding the right concepts
    - provides the simplest model for a program/system
  - Generalization
    - Across various contexts – eg, data, processing etc
  - Separation of interface and implementation
- Benefits:
  - Easier to understandable
  - Easier to implement and test
  - Provides more reusable modules

Information Hiding

- Key element in Interface Abstraction
- Separates interface from implementation
- The practice of hiding the details of a module with the goal of controlling access to the details from the rest of the system.
- Enables concurrent development if define interfaces first
- A programmer can concentrate on one module at a time.
- Reduces (inter)dependencies

Stepwise Refinement

- Generally a top-down approach
- A problem is approached in stages.
  - Similar steps are followed during each stage
  - Primary difference being the level of detail involved.
- Move from very abstract description to more detailed descriptions
  - Data
    - Data models
    - Data structures
    - Data representations
  - Processing
    - High level pseudo-code often a useful mechanism
    - Functional decomposition
      - High level main program
      - Hierarchical structure – the deeper the more detail

Virtual Machine

- Generally a bottom up approach
- Begin with basic machine
  - Basic operations
  - Basic elementary data
- Add layers of abstractions
  - Higher level operations
  - Richer and more complex data
- Until you reach the appropriate application layer
  - Right abstractions for the application
  - Simple to understand operations and data

Measurement & Evaluation: Reviews

- Basic Premise:
  - Fresh look at code or document
- Advantages:
  - Easier and cheaper to catch faults as early as possible
  - Hard faults: ones that compromise system functionality
    - Critical functional requirements
    - Critical non-functional requirements
      - Performance, reliability, etc.
  - Soft faults: ones that compromise maintainability
    - Coding style
    - Architectural structure

Measurement & Evaluation: Testing

- Determine ranges or classes of inputs.
- Determine the expected behavior of the program for various inputs
  - Called "the Oracle"
- Run the program and observe the resulting behavior.
- Compare the "test results" with "the oracle"
- Revise Tests and System until
  - Satisfied
  - Sufficient "coverage" of
    - Requirements (black box testing)
    - Implementation (white box testing)

EE 322C
Data Structures

Lecture 12

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm – ACES 5.104

Announcements

- Exam 1 – Next Wednesday, 5-6:15 (NOT AM ;-)
- Homework Assignment 2
- Topics of the day
  - Array review

Arrays

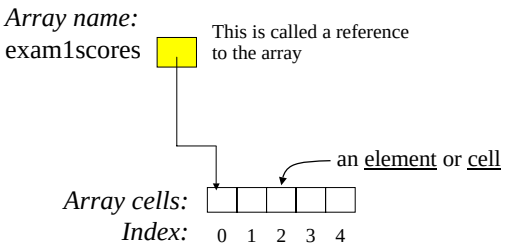
- An array is *sequence* of variables of the **same** data type. The values can be all: int, double, char, boolean, string, etc.
- An array is a composite data structure/object (like string) rather than a primitive data type (like int).
- The whole collection is referred to by a single array name
- The individual variables (sometimes called elements or cells) are accessed by using an integer index (also called a subscript) which indicates that value's position in the set.
- Index values range from 0 to one less than the number of elements

Arrays

- The length attribute of an array refers to the number of elements allocated to the array - fixed at allocation time
- An array allows us to store and process all the values in a collection (or set) of data, rather than just the most recent value. They are stored in consecutive memory locations.
  - e.g. what if we wanted to store and manipulate in various ways **all of the EE322C exam 1 scores for this semester**

Arrays in Memory

Example - a set of 5 EE322C exam 1 scores

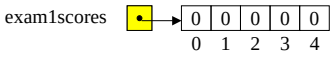


Creating an Array

Declaration statement syntax is:

```
basetype arrayname [integer expression];
int exam1scores[5];
```

Sets up a structure in memory like:

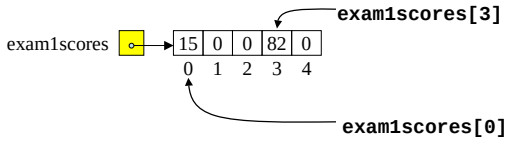


- Array elements contain default values: 0 for numerics, false for booleans, null for chars, strings, etc.
- An array element is a full fledged variable

Referring to array elements

array reference syntax: *arrayname* [*index*]
index can be a value or an expression, e.g.

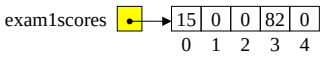
```
int exam1scores [5];
exam1scores[0] = 15;
exam1scores[3] = 82;
exam1scores[4] = exam1scores[2];
```



Referring to array elements

array reference syntax: *arrayname* [*index*]
index can be a value or an expression, e.g.

```
int exam1scores [5];
exam1scores[0] = 15;
exam1scores[3] = 82;
exam1scores[4] = exam1scores[2];
```



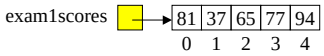
A value from the array can be used anywhere a variable of the base type can be used. E.g.

```
int x = 2 * exam1scores[3]
exam1scores[exam1scores[2]];
```

What's the value of x?

Initializing Array Elements

```
// reading values in for example
const int NUMBEROFSTUDENTS = 5;
int exam1scores [NUMBEROFSTUDENTS];
for (int i = 0; i < NUMBEROFSTUDENTS; i++)
   cin >> exam1scores[i];
```




```
Const int NUMBEROFSTUDENTS = 5;
int exam1scores [NUMBEROFSTUDENTS];
for (int i = 1; i <= NUMBEROFSTUDENTS; i++)
   cin >> exam1scores[i];
```

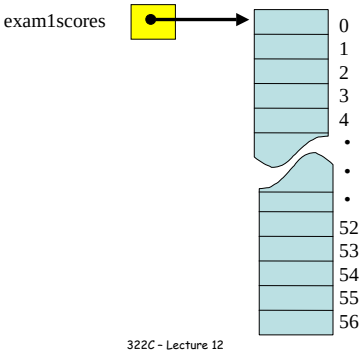
What is wrong with this code?
It should Lead to an Error! But doesn't!

Exam 1 Scores Example

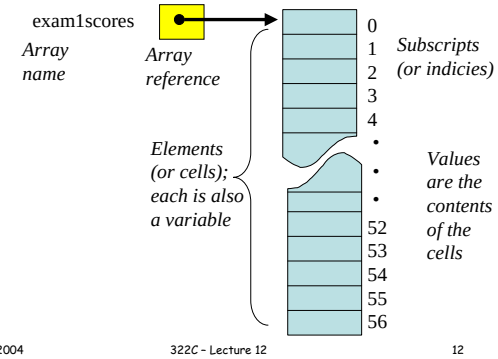
Declaring the array

```
const  int NUMBEROFSTUDENTS = 57;
int exam1scores [NUMBEROFSTUDENTS];
```


Represented as a Sequence of `ints`



Represented as a Sequence of `ints`



Exam 1 Scores Example

```
const int NUMBEROFSTUDENTS = 57;
int exam1scores  [NUMBEROFSTUDENTS];
// input all the exam scores from the keyboard
for (int i = 0; i < NUMBEROFSTUDENTS; i++)
        {cin >> exam1scores[i];}
```

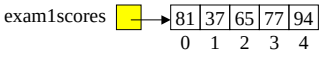

Exam 1 Scores Example

```
const int NUMBEROFSTUDENTS = 57;
int exam1scores  [NUMBEROFSTUDENTS];
// input all the exam scores from the keyboard
for (int i = 0; i < NUMBEROFSTUDENTS; i++)
            {cin >> exam1scores[i];}

// find the class  average of the exam 1 scores
   double average, sum = 0;
   for (int i = 0; i < NUMBEROFSTUDENTS; i++)
    { sum = sum + exam1scores [i]; // the running sum
}
    average =  sum / NUMBEROFSTUDENTS;
    cout << "the class average is: " << average ;
```


Using Initializer Lists

Another way of creating and initializing an array:

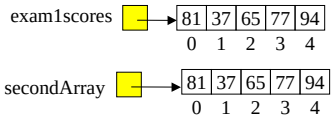
```
int myints = {0, 1, 2, 3, 4};
double mydoubles = {23.1, 0.34, 22};
string mystrings = {"Hi", "there"};
int exam1scores = {81, 37, 65, 77, 94};
```



- Can only use in the declaration statement. The length is figured by the number of values.
- If you already know the values and the set is small and you want the values hard coded into the program then do it this way

Making a Copy of an Array

```
const int NUMBEROFSTUDENTS = 57;
int exam1scores [NUMBEROFSTUDENTS];
//code to fill exam1scores would be here
int secondArray [NUMBEROFSTUDENTS];

for(int index =0; index < NUMBEROFSTUDENTS; index++)
      secondArray[index] = exam1scores[index];
```



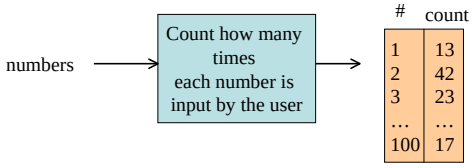

```
secondArray = exam1scores; //doesn't make a copy
```


Counters Example

- **Problem** - keep track of how many times a given number in the range from 1 to 100 was input by the user over a series of input numbers.

Counters Example

- **Problem** - keep track of how many times a given number in the range from 1 to 100 was input by the user over a series of input numbers.



Solution ideas - let's use a 100 element array of counters. What's the correspondence between input values and array elements ?

Counters Example

- **Problem** - keep track of how many times a given number in the range from 1 to 100 was input by the user over a series of input numbers.

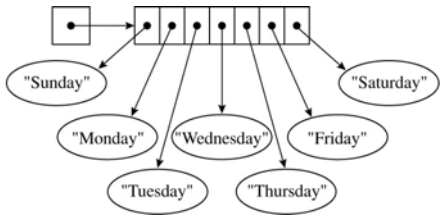
```
. . .
int  counters [100];  // declare, allocate&initialize array
int number;
const int SENTINEL = 999;
cin >> number;
while (number ! = SENTINEL ) // repeatedly input & process #s
{   // increment the counter for that number, read the next number
    counters [number - 1] = counters [number -1]  + 1;
    cin >> number;
}
// output how many of each number was input
for (int i =1; i <= 100; i++)
  cout << "the counter for number " << i << " is: " <<
      counters [i - 1]);
```


Initializing an Array of Strings

```
const string DAYS_OF_WEEK  =
   { "Sunday",
     "Monday",
     "Tuesday",
     "Wednesday",
     "Thursday",
     "Friday",
     "Saturday"
    };
```


How It Is Stored

DAYS_OF_WEEK



Encapsulate that into a function

```
// a function (procedure) that prints the day of week
// given a day number from 1 to 7
void printDayName (int day)
{  const string DAYS_OF_WEEK  =
     { "Sunday", "Monday", "Tuesday", "Wednesday",
       "Thursday", "Friday", "Saturday"
     };
     cout << DAYS_OF_WEEK [day – 1];
}
// Called from the main or other function like:
printDayName (3);  // what is output?
```


Partially Filled Array

- Arrays cannot grow after they have been allocated
- You can allocate more space (i.e. extra) than you believe your application will need
- If you guess too low, you will still run out of space and terminate
- You do not need to use all the elements in an array (but total computer memory is finite)

Partially Filled Arrays - Ex.

- What if some students dropped before exam 1? or what if I want to use this program again next semester when I will have more or fewer students?
- In this case I might wish to treat 100 as a maximum capacity but not necessarily the actual number of exam scores - so the array will be partially filled
- Here is an example of how we initialize the values

```
const int MAX_STUDENTS = 100;
int exam1scores [MAX_STUDENTS ];
int arraysize;
cin >> arraysize; // read in the actual size
for (int i = 0; i < arraysize; i++)
   cin >> exam1scores[i]; // read in values
```

- Note - arraysize is a variable that you have to keep track of

EE 322C
Data Structures

Lecture 2

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

Announcements

- Exam 1 – Wednesday 5 – 6:15pm
- Review:
- Today:
  - Exam Rules and Topics
  - Wrap up Arrays

Exam Rules and Expectations

- Exam will start at exactly 5PM and finish at exactly 6:15PM on Monday
- Bring one or more pencils
- No other devices are allowed
- No questions answered during the exam

```
if (confused) then
{
  make an assumption
  write it down
  proceed to answer the question on that basis
}
```

- Prepare - eg, use cppreference.com and the other links provided on the web site.

Topic Areas

- Software engineering principles
  - Design techniques (OO vs functional), ADTs, algorithms
- C++ language
  - Compiler, built-in features, function library, class library, preprocessor
- Simple data structures
  - string, etc.
- Defining new and richer data types
  - Safe array, bank accounts, etc

Topics Covered (1)

- What is SWE, programming, SWE tracks in ECE, etc.
- How to think about and solve programming problems
- Modularity, encapsulation and abstraction
- Step-wise refinement, virtual machine
- OO thinking
- C and C++ basics (EE 312 topics assumed)
- Strings
- Building your own data types
- Booleans
- Functions
- Overloaded functions
- Structs, Bit fields, Unions. Enumerations, Typedef

Topics Covered (2)

- Classes and objects
- Construction/destruction of objects
- Subclasses and Inheritance
- Stream and File I/O
- Arrays of objects
- Pointers
- References
- Dynamic memory allocation
- Polymorphism & Overloaded operators
- Exception handling

Simple Array Algorithms

- Once you have an array with values in it there are a few common algorithms (and variants) that are often used in handling arrays
- Ordered Vs Unordered arrays - it depends
  - Counting the number of values
  - Finding a given value
  - Removing a value
  - Inserting a value
  - Sorting an unordered array into order

Counting Value Occurrences

```
const int NRSTUDENTS = 100;
int exam1scores [NRSTUDENTS];
// code to fill in exam1scores in here
int targetValue = ... ;
  // fill in the value to be counted
int count = 0;
for(i = 0; i < NRSTUDENTS; i++)
  {
      if (exam1scores [i] == targetValue)
        count = count + 1;
  }
cout << count  << " matches were found";
```


Searching For a Given Value

```
const int NRSTUDENTS = 100;
int exam1scores [NRSTUDENTS];
// code to fill in exam1scores in here
int targetScore =  90 ;
int i = 0;
bool found = false;
//  This loop terminates as soon as it finds the value 90
while (i < NRSTUDENTS && !found)
   {  if (exam1scores [i] == targetScore)
         found = true;
      else
         i = i + 1;
   }
if (found)
   cout << "Item was found at index " <<  i;
      // or do other stuff with the found item
```


Adding and Removing Values

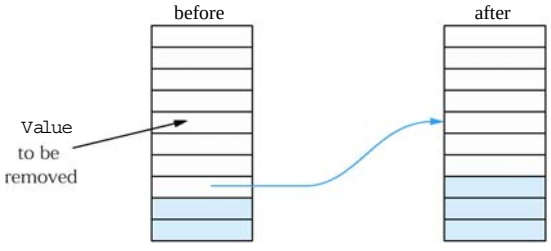
- For unordered arrays you can just add the new value on the end, e.g. (assume that the variable **last** holds the index value of the end element in the array)

```
exam1scores [last +1] = newValue;
last++;
```

- For unordered arrays you can replace the value to be removed with the last value

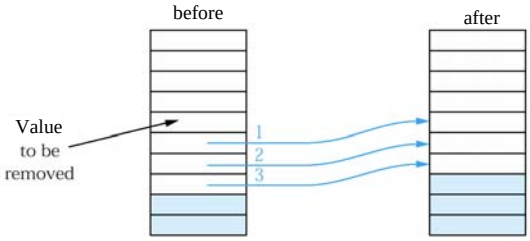
```
exam1scores [removalindex] = exam1scores [last];
last--;
```


Removing a Value from an Unordered Array



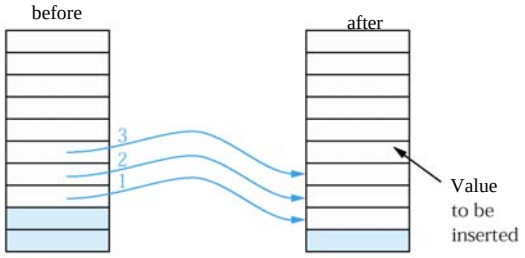
Take the last value and use it to replace the value to be removed
The array now has one less value in it

Removing a Value from an Unordered Array



All subsequent values must move up one slot

Inserting a Value into an Ordered Array

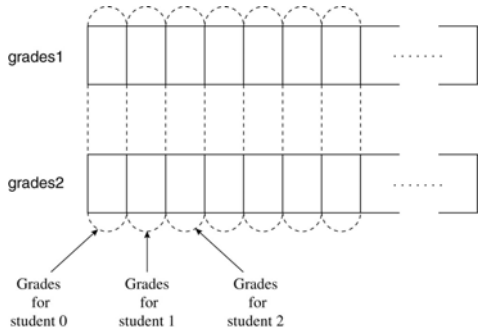


All subsequent values must be moved down to make room
For an unordered array, just stick the new value at the end

Parallel Arrays

- This loop counts the number of students whose performance improved from the first exam to the second exam

```
int grades1 [NRSTUDENTS];
int grades2 [NRSTUDENTS];
// assume the two arrays get values in here
int improved = 0;
for (int i = 0; i < NRSTUDENTS; i++)
{
  if (grades1[i] < grades2[i]) improved++;
}
```

Parallel Arrays

- of different types -

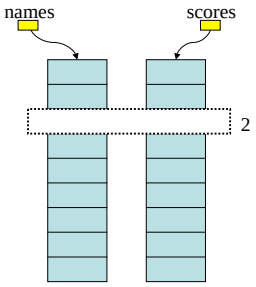
- Tabular data to be stored

Name	Score
Abe	71
Joe	64
Maximus	72
Summi	98
…	…

- A solution: 2 "parallel" arrays of different types

```
string names;
int scores;
```

Or better use an array of structures



Corresponding elements must stay in correspondence by matching index

Simple Sorting

- **Goal** - Change an unordered array into an ordered array. The array will now have the values sorted in either ascending or descending numeric order as required.
- How can that be done?

Ideas to get started?

1. Find the smallest value in the array and swap it with the 0th element in the array.
2. ….

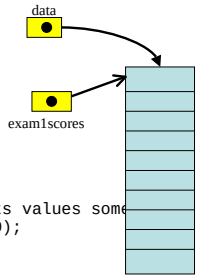
Now what?

Selection Sort - Algorithm

- **Goal - Change an unordered array into an ordered array.**
- The array will now have the values sorted in ascending numeric order. There are many ways to do this.
- Given an unordered array named A of length n, for example:
  1. Find the smallest element among the elements A[0]..A[n-1] and identify it as A[min]
  2. Swap the values of A[0] and A[min] so A[0] contains the smallest element, now A[1]..A[n-1] are not sorted
  3. Next find the smallest element among the remaining elements A[1]..A[n-1] and call it A[min]
  4. Swap A[1] and A[min] so A[1] contains the second smallest element, now A[2]..A[n-1] are not sorted
  5. Repeat this process similarly starting at A[2], A[3], and so on until the final element is reached

Passing an Array as a Parameter

```
/* average is a function that computes and returns the  average
 of the values in a given (passed) integer array*/
 double average (int data[ ], LENGTH)
{  if (LENGTH == 0) return 0;
   double sum = 0;
   for (int i = 0; i < LENGTH; i++)
      sum = sum + data[i];
   return sum / LENGTH;
}
```



- Called from another function as e.g.

```
double  avg;
int exam1scores [100];
// assume that the array gets its values some
avg  = average (exam1scores, 100);
```

- Call by reference to the array

Returning an Array

- `/* This function constructs and returns an integer array with random values, each in the range from 0 to n-1 */`

```
int[ ] randomData (int size, int n)
{
   int data [size];
   for (int i = 0; i < size; i++)
      data[i] = rand ( )% n;
   return data;
}
```

- Called in the main function for example as

```
int examScores = randomData (classSize, 101);
```


Selection Sort - Algorithm

- **Goal - Change an unordered array into an ordered array.**
- The array will then have the values sorted in ascending numeric order. There are many ways to do this.
- Given an unordered array named A of length n, for example:
  1. Find the smallest value among the elements A[0]..A[n-1] and identify it's position as A[min]
  2. Swap the values of A[0] and A[min] so A[0] now contains the smallest value, now A[1]..A[n-1] are not sorted
  3. Next find the smallest value among the remaining elements A[1]..A[n-1] and identify it's position as A[min]
  4. Swap A[1] and A[min] so A[1] contains the second smallest value, now A[2]..A[n-1] are not sorted
  5. Repeat this process similarly starting at A[2], A[3], and so on until the final element is reached. The array is now sorted.

Exercise

- Change the algorithm so that it sorts into descending numerical order

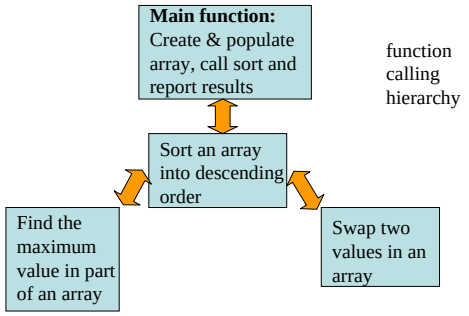
Selection Sort - Algorithm(2)

- **Goal - Change an unordered array into an ordered array.**
- The array will then have the values sorted in **descending** numeric order. There are many ways to do this.
- Given an unordered array named A of length n, for example:
  1. Find the largest value among the elements A[0]..A[n-1] and identify it's position as A[max]
  2. Swap the values of A[0] and A[max] so A[0] now contains the largest value, now A[1]..A[n-1] are not sorted
  3. Next find the largest value among the remaining elements A[1]..A[n-1] and identify it's position as A[max]
  4. Swap A[1] and A[max] so A[1] contains the second largest value, now A[2]..A[n-1] are not sorted
  5. Repeat this process similarly starting at A[2], A[3], and so on until the final element is reached. The array is now sorted.

Exercise (cont.)

- Now inspect the algorithm looking for potential sub-processes that have to be done more than once
- These become functions/procedures in a modular solution program
- What are they?

Modular Structure of a Well Formed Program



SelectionSort – part 1

```
// usual stuff up here
int main ( )
{ const int MAXSIZE = . . .;  int arraysize;
  double myArray [MAXSIZE];
    // assume that myArray values are partially filled in here
  selectionSort (myArray, arraysize); //call the sort function
    // other stuff here
  return 0;
}
void selectionSort (double array [ ], int size)
{ // this function sorts the given array into descending order
  int max; // will hold the index of the largest value found
  for (int i=0; i < size; i++)
     {
       max = findMaximum (array, i, size);
       swap(array, i, max);
     }
 }
```


SelectionSort – part 2

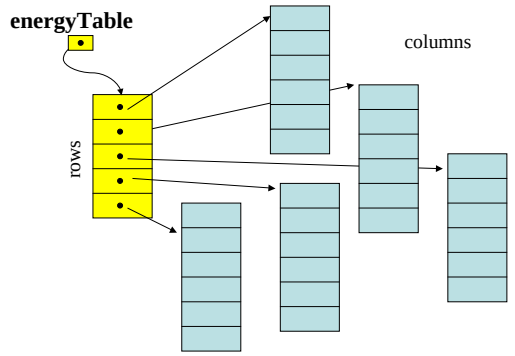
```
int findMaximum (double array [ ], int i, int size)
{// function returns the index of the largest value in array
    int j, max = i;
    for (j= i + 1; j < size; j++)
        if (array[j] > array[max]) max =j;
    return max;
}
void swap (double array [ ], int i, int j)
{//function swaps the 2 values at positions i and j in array
    double temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}
```


Two-Dimensional Arrays

- Used for tables and matrices
- Declaration similar to one dimensional arrays
- Need to specify both the number of rows and columns during allocation
- Example:

```
const int
ROWS = 5,
COLS = 6;
double energyTable[ROWS][COLS];
```


A 2D Array in memory



Energy Table

Years	Coal (0)	Gas (1)	Oil (2)	Hydro (3)	Nuclear (4)	Other (5)	
0	18.9	19.4	34.2	2.9	5.7	0.3	1989
1	19.1	19.3	33.6	3.0	6.2	0.2	1990
2	18.8	19.6	32.9	3.1	6.6	0.2	1991
3	18.9	20.3	33.5	2.8	6.7	0.2	1992
4	19.6	20.8	33.8	3.1	6.5	0.2	1993

Energy Sources

Nested for Loops

- Nested loops are frequently used to process two-dimensional arrays
- Often the body of inner loop is where the main computation is done
- Example:

```
for (i = 0; i < ROWS;  i++)
{  // stuff before the inner loop
  for (j = 0; j < COLS; j++)
   { // body of inner loop operates on the elements
     }
  // stuff after the  inner loop
}
```

By convention we usually refer to the specific elements as *tablename* [i, j], where i is the row index and j is the column index

Reading in values for EnergyTable

```
const int ROWS = 5, COLS = 6;
double energyTable [ROWS][COLS];
int i, j;
// reads 30 numbers needed to fill up the entries in the
// energyTable one row at a time
for (i = 0; i < ROWS; i++)
  for (j = 0; j < COLS; j++)
     cin >> energyTable[i][j];
```


OR Using an Initializer List

```
double energyTable =
  {
    {18.9, 19.4, 34.2, 3.9, 5.7, 0.3},
    {19.1, 19.3, 33.6, 3.0, 6.2, 0.2},
    {18.8, 19.6, 32.9, 3.1, 6.6, 0.2},
    {18.9, 20.3, 33.5, 2.8, 6.7, 0.2},
    {19.6, 20.8, 33.8, 3.1, 6.5, 0.2}
  };
```


Computing Year (Row) Totals

```
double  yearTotals [ROWS];
for (i = 0; i < ROWS; i++)
{ // compute total for each year i
   yearTotals[i] = 0.0;
   for (j =0; j < COLS; j++)
     yearTotals[i] =
        yearTotals[i] + energyTable [i][j];
}
```


Computing Column Totals

```
double  sourceTotals [COLS];
```

After you have the row and column totals, then output the energy table with the totals added to it and also with descriptive labels on the table

Resultant Output Table With Labels and Totals

Terminal Window

	Coal	Gas	Oil	Hydro	Nuclear	Other	Total
1989	18.9	19.4	34.2	2.9	5.7	0.3	81.4
1990	19.1	19.3	33.6	3.0	6.2	0.2	81.4
1991	18.8	19.6	32.9	3.1	6.6	0.2	81.2
1992	18.9	20.3	33.5	2.8	6.7	0.2	82.4
1993	19.6	20.8	33.8	3.1	6.5	0.2	84.0
Total	95.3	99.4	168.0	14.9	31.7	1.1	410.4

EE 322C
Data Structures

Lecture 14

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

STL

- The Standard Template Library, or STL, is a C++ library which provides many of the basic data structures and algorithms.
- The STL is a generic library, that is, almost every component in the STL is a template.
- It consists of:
  - ✓Container classes
  - ✓Algorithms
  - ✓Iterators
  - ✓Adaptors
- And a set of other special components:
  - ✓Function Objects
  - ✓Allocators
  - ✓Predicates
  - ✓Comparison functions

STL

- STL started in the 1970s by Stepanov/Lee at SGI/HP and was accepted into C++ by the ANSI/ISO C++ standards committee in 1994.
- Programming Guide: http://www.sgi.com/tech/stl/index.html

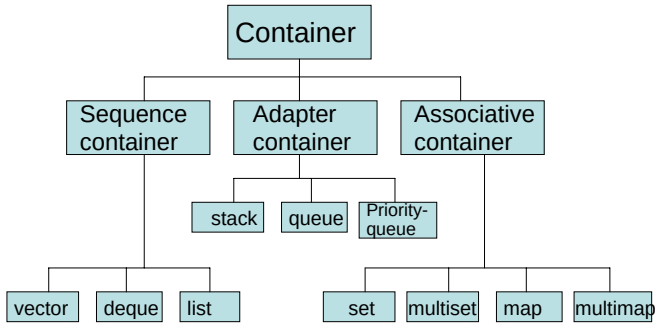
Containers

- A container is a generic data structure that stores a large collection of elements. It has common operations for adding, removing and accessing elements.
- The STL provides 10 container classes for solving a wide range of problems.
- The elements do not have to be stored in any definite order for a given Container type.
- A container owns it's elements and they are deallocated when a Container object is destroyed.
- STL containers are very close to the efficiency of hand-coded, type-specific containers.
- An iterator is an integral type associated with Containers that can be used to to iterate (cycle) through the Container's elements.

Container Types

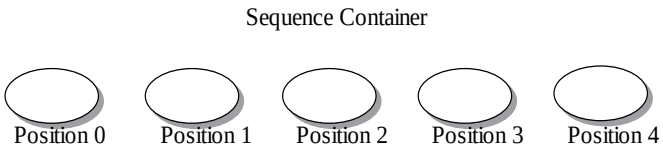
Sequence Containers	Adapter Containers	Associative Containers
Vector	Stack	Set, Multiset
Deque	Queue	Map, Mutltimap
List	Priority Queue	

Container Class Hierarchy



Sequence Containers

A sequence container stores data by position in linear order Oth element, 1st element, 2nd element, and so forth.



Sequence Containers

- The array data structure is a **sequence** container.
  - ✓ It defines a block of consecutive data values of the same type.
  - ✓ Arrays are direct access containers.
  - ✓ An index may be used to select any item in the list without referencing any of the other items
- The vector **sequence** container provides direct access through an index and grows dynamically at the rear as needed.
  - ✓ Insertion and deletion at the rear of the sequence is very efficient - these operations inside a vector are not efficient
- The list **sequence** container stores elements by position.
  - ✓ List containers do not permit direct access
    - must start at the first position (front) and move from element to element until you locate the data value.
  - ✓ The power of a list container is its ability to efficiently add and remove items at any position in the sequence.

Adapter Containers

- An adapter contains a sequence container as its underlying storage structure.
- The programmer interface for an adapter provides only a restricted set of operations from the underlying storage structure.

Adapter Containers

- Stacks and a queues are adapter containers that restrict how elements enter and leave a sequence.
  - ✓ A stack allows insertion and access at only one end of the sequence, called the top.
  - ✓ A queue is a container that allows access only at the front and insertion at the rear of the sequence.
- Similar to a stack or queue, the priority queue adapter container restricts access operations.
  - ✓ Elements have a priority associated with them
  - ✓ Elements can enter the priority queue in any order.
  - ✓ Once in the container, only the highest priority element may be accessed.

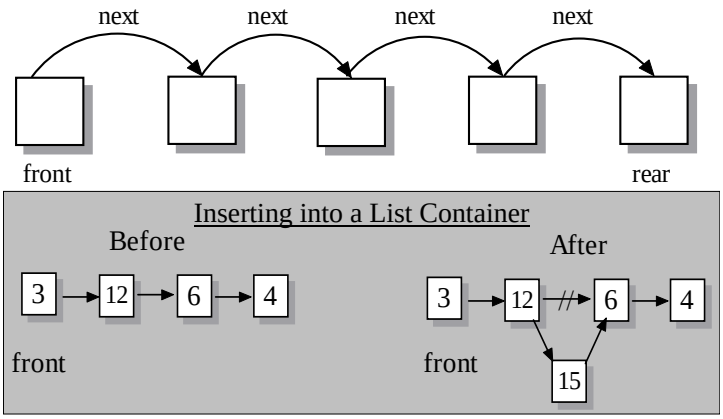
Associative Containers

- Associative containers store elements by key.
  - Ex: name, social security number, or part number.
- A program accesses an element in an associative container by its key, which may bear no relationship to the location of the element in the container.

Associative Containers

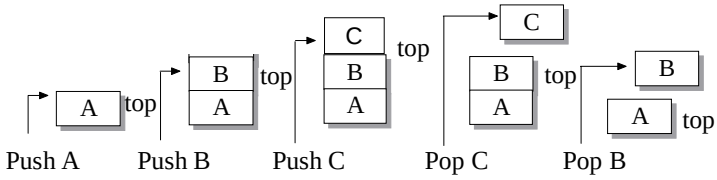
- A set is a collection of unique values, called keys or set members.
  - ✓ The set container has a series of operations that allow a programmer to determine if an item is a member of the set and to very efficiently insert and delete items.
- A map is a storage structure that allows a programmer to use a key as an index to the data.
  - ✓ Maps do not store data by position and instead use key-access to data allowing a programmer to treat them as though they were a vector or array.

The List Container



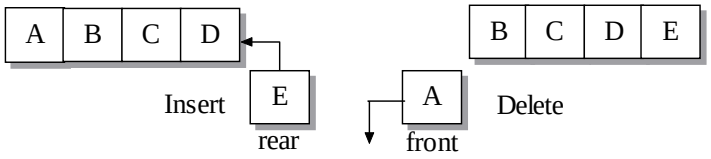
Stack Containers

A stack allows access at only one end of the sequence, called the top.



Queue Containers

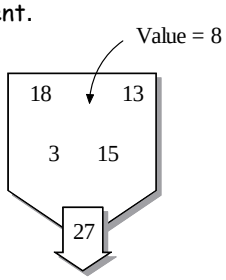
A queue is a container that allows access only at the front and rear of the sequence.



Priority Queue Containers

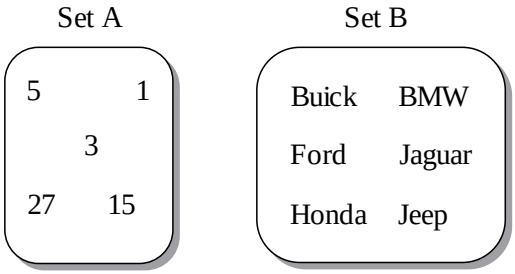
A priority queue is a storage structure that has restricted access operations similar to a stack or queue.

Elements can enter the priority queue in any order. Once in the container, a delete operation removes the highest (or lowest) priority element.



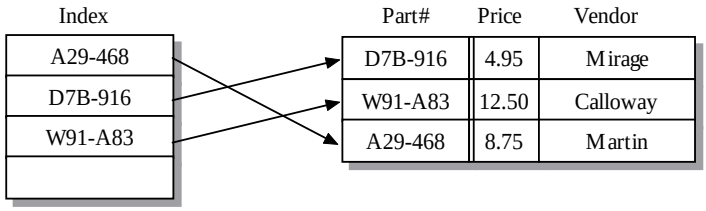
Set Containers

A set is a collection of unique values, called keys or set members.



Map Containers

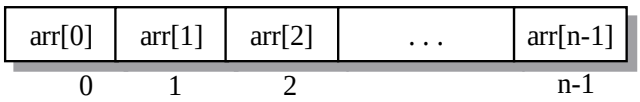
A map is a storage structure that implements a key-value relationship.



C++ Arrays - Again

An array is a fixed-size collection of values of the same data type.

An array is a container that stores the n (size) elements in a contiguous block of memory.



Evaluating an Array as a Container

- The size of an array is fixed at the time of its declaration and cannot be changed during the runtime.
  - An array cannot report its size. A separate integer variable is required in order to keep track of its size.
- C++ arrays do not allow the assignment of one array to another.
  - The copying of an array requires the generation of a loop structure with the array size as an upper bound.

Program Analysis Questions

- Does it do what I want it to do?
- Does it work correctly according to the requirements given?
- Does the documentation describe how to use it and how it works?
- Is the code of good style?
- Is the program well-designed?
  - Good modularity
  - Useful encapsulation
  - The right abstractions
- **How efficient is the program?**

Selection Sort Function

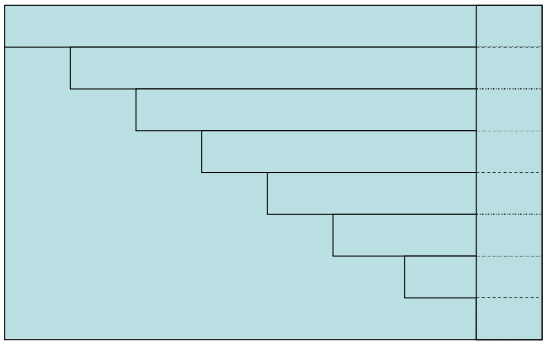
```
void selectionSort(int array[ ], int n)
{    int smallIndex;// index of the smallest element in the (sub) array
   int pass, j;
   int temp;
   // pass has the range 0 to n-2
      for (pass = 0; pass < n-1; pass++)
   {     // scan the sublist starting at index pass
         smallIndex = pass;
         // j traverses the sublist array[pass+1] to array[n-1]
         for (j = pass+1; j < n; j++)
          // if smaller element found, assign smallIndex to that position
         { if (array[j] < array[smallIndex]) smallIndex = j;}
         // if smallIndex and pass are not the same location, swap the
         // smallest item in the sublist with array[pass]
         if (smallIndex != pass)
         {  temp = array[pass];
            array[pass] = array[smallIndex];
            array[smallIndex] = temp;
         }
   }
}
```

 How efficient is this ?

Performance Analysis

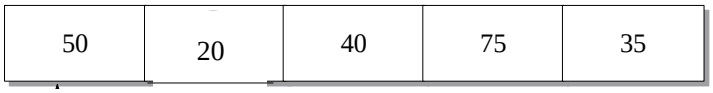
- Performance evaluation questions:
  - How much computing time does a certain algorithm take?
  - How much storage (memory) does a certain algorithm use?
  - A priori estimates can be made
  - A posteriori measurements can be taken
- Big-O notation measures the efficiency of an algorithm by estimating the number of key operations that the algorithm must perform. It provides an order of magnitude estimate.
  - For searching and sorting algorithms, the operation is data comparison.
  - Big-O measure is very useful for selecting among competing algorithms.
  - Timing data obtained from a program provides experimental evidence to support the greater efficiency claims
- For selection sort how many comparisons have to be made?
  - Can you estimate that?
  - What does it depend on?

Selection Sort



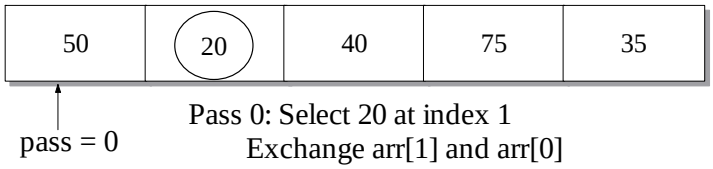
Selection Sort- 5 Element Array

- Start with the values {50, 20, 40, 75, 35}
- We want to sort the values into ascending order



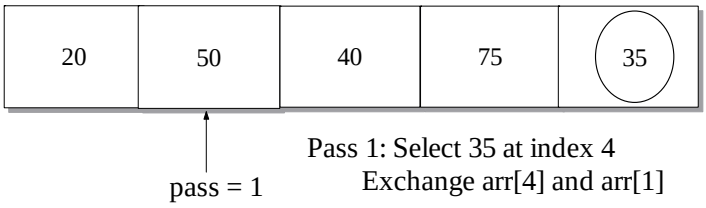
Selection Sort - 5 Element Array

- Pass 0:
- Scan the entire list from arr[0] to arr[4] and identify 20 at index 1 as the smallest element.
- Exchange 20 with arr[0] = 50, the first element in the list.



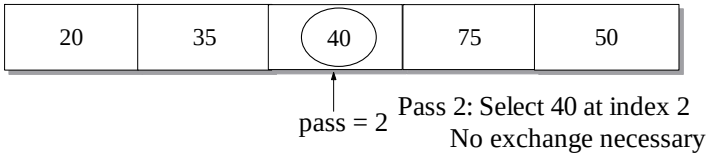
Selection Sort - 5 Element Array

- Pass 1:
- Scan the sublist 50, 40, 75, and 35.
- Exchange the smallest element 35 at index 4 with arr[1] = 50.



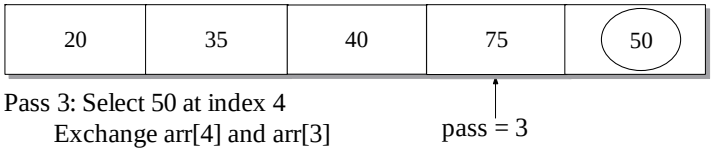
Selection Sort - 5 Element Array

- Pass 2:
- Locate the smallest element in the sublist 40, 75, and 50.

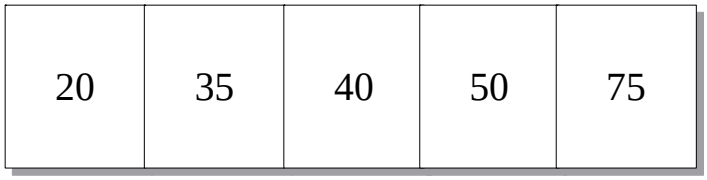


Selection Sort - 5 Element Array

- Pass 3:
- Two elements remain to be sorted.
- Scan the sublist 75, 50 and exchange the smaller element with arr[3].
- The exchange places 50 at index 4 in arr[3].



Selection Sort - 5 Element Array



Sorted list

Big-O notation

- Summary of 5 element array
  - Pass 0: 4 comparisons max
  - Pass 1: 3 comparisons max
  - Pass 2: 2 comarisons max
  - Pass 3: 1 comparison max
  - Hence, 10 comparisons = ((n*n)-n)/2 = (5*5)-5)/2 = 10
- For the selection sort, the number of comparisons isT(n) ~= $O(n^2-n/2)$.
- For n = 100: T(100) = 100^2-100/2 = 10000-100/2 = 9900/2 = 4950
- Entire expression is called the "Big-O" measure for the algorithm.

EE 322C
Data Structures

Lecture 15

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

Exam Results

- Curved – slightly skewed towards lower grades
  - Median = 75.5
  - Mean = 75.183
- Score Ranges
  - A 81-91 15
  - B 73-80 21
  - C 60-72 22
  - D 50-60 2
  - F <50 0
- Improvement will count
- Will make the best case for you overall

Exam Results - Overall

- 91 *
- 89.5 ****
- 87 **
- 86 *
- 83 **
- 82.5 **
- 82 **
- 81 *
- ---
- 80 *
- 79.5 ****
- 79 **
- 78.5 *
- 77.5 ***
- 77 *
- 76.5 **
- 75.5 **
- 74 **
- 73.5 **
- ---
- 72.5 *
- 72 *
- 71.5 ***
- 71 ****
- 70.5 *
- 70 **
- 68.5 **
- 68 ***
- 64.5 *
- 64 ***
- 63.5 *
- 61.5 *
- ---
- 58.5 *
- 56 *

Exam Results: T/F

- Overall, pretty good – median=16, mean=14.73
  - 18 *****
  - 16 ***** ***** ***** ***** *****
  - 14 ***** ***** ***** ***
  - 12 ***** *****
  - 10 *
  - 8
  - 6
  - 4
  - 2
  - 0

Exam Results: Multiple Choice

- 21 points - median=17.5, mean=17.32
  - 21 **
  - 20.5 *
  - 20 *****
  - 19.5 ***
  - 19 ****
  - 18.5 ***** *
  - 18 ***** ***
  - 17.5 ***** *
  - 17 ***** *
  - 16.5 ****
  - 16 *
  - 15.5 **
  - 15 ***
  - 14.5 ****
  - 14 **
  - 13 *
  - 12.5 *
  - 10.5 *

Exam Results – Fill In

- Wider spread – median=18.5 , mean=18.87
  - 24 ***** ***** **
  - 22.5 **
  - 21 ***** ***** ****
  - 19 *
  - 18 ***** ***** ***
  - 16.5 **
  - 15 ***** *****
  - 12 *****
  - 9 *

Exam Results – What's Wrong

- 12 points – median=9, mean=8.38,
  - 12 ****
  - 11 ***
  - 10.5 ***
  - 10 *
  - 9.5 *
  - 9 ***** ***** ***** ***** ***
  - 8 *****
  - 7.5 ****
  - 7 ****
  - 6 ***** ****
  - 5 *
  - 4 *
  - 3 *

Exam Results - Evaluate

- 9 points – median=6.75, mean=6.275
  - 9 ***
  - 8.5 **
  - 8 ***** ***
  - 7.5 ***** *
  - 7 ***** *****
  - 6.5 ***** *
  - 6 ***** *
  - 5.5 ***
  - 5 **
  - 4.5 *****
  - 4 **
  - 3.5 **
  - 3 *****
  - 2.5 *

Exam Results - Debug

- 8 points – median=4, mean=4.24
  - 8 ***
  - 7.5 *
  - 7 *****
  - 6 *****
  - 5.5 **
  - 5 ***** *
  - 4.5 ****
  - 4 ***** **
  - 3.5 ****
  - 3 ***** ***** ***
  - 2.5 **
  - 2 ****
  - 1.5 **
  - 1 **

Exam Results - Complete

- 8 points – median=5.5, mean=5.37
  - 9 ****
  - 8.5 **
  - 8 ***** *
  - 7.5 **
  - 7 ***** *
  - 6.5 **
  - 6 ****
  - 5.5 ****
  - 5 *****
  - 4.5 ***
  - 4 *****
  - 3.5 ****
  - 3 *****
  - 2.5 **
  - 2 **
  - 1 *
  - 0 **

What is Recursion?

- ***Recursion*** is when a function calls itself
- It is an important problem solving approach in CS. Problems that are amenable to recursive solutions have:
  - One or more stopping cases with a simple, nonrecursive solutions
  - The other cases can be reduced to simpler problems closer to the stopping cases
  - Eventually the problem can be reduced to simple stopping cases
- The classic example is computing the factorial of a non-negative integer. The mathematical formulation of n! is:
  - 0! = 1 by definition
  - n! = n * (n-1)! Recursively defined

N!

- *Represents the number of permutations of n* symbols, e.g. the three symbols a,c,r
  - 3! = 6: car, rac, arc, acr, rca, cra
- n! = (n-1)! × n (recursively)
- 0! = 1
  1! = 1
  2! = 2
  3! = 6
  4! = 24
  5! = 120

N-Factorial Program

```
// Compute factorial of a given integer
// usual stuff in here
int  factorial (int);
int main ( )
{
   cout.<< "Please enter a nonnegative integer";
   int num;
   cin >> num;
   cout << "The factorial of " << num << " is " <<
   factorial(num);
   return 0;
  }
// Recursive function for computing the factorial of an
// integer n
  int  factorial (int n)
  {
       if (n == 0) return 1;  // terminating case
       else return n*factorial(n-1);
         // Call factorial function recursively for n-1
  }
```


Recursive Descent and Ascent

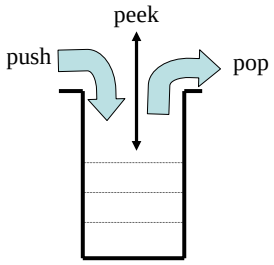
- Every recursive call is like a call to a **new copy** of the function
- Every recursive call must simplify the computation in some way
- There must be special cases to handle the simplest computations (terminal cases)
- factorial(4) calls factorial(3)
  factorial(3) calls factorial(2)
  factorial(2) calls factorial(1)
  factorial(1) calls factorial(0)
  factorial(0) returns 1
  factorial(1) returns 1 * 1 = 1
  factorial(2) returns 1 * 2 = 2
  factorial(3) returns 2 * 3 = 6
  factorial(4) returns 6 * 4 = 24

Recursion uses the Stack

- C++ has an internal data structure called the program stack that is used to store local variable information

Recursion uses the Stack

- C++ has an internal data structure called the program stack that is uses to store local variable information on



A Stack enforces a LIFO discipline (last in, first out)

Computing Fibonacci Numbers

- Fibonacci numbers form a sequence of integers in which each number in the sequence is the sum of the two preceding numbers.
- The first two numbers in the sequence have the values 0 and 1.
- Find Fibonacci numbers using recursion.
  fib (0) = 0;
  fib (1) =1;
  fib(n) = fib(n-2) + fib(n-1); for n >= 2

Fibonacci Program

```
// usual stuff here
int fib (int)
int main( )
{
   cout << "Enter a non-negative integer n: ";
   int num;
   cin >> num;
   cout << "the fibonacci number of " <<  num << " = " <<  fib(num);
   return 0;
}
// Computes the value of the nth Fibonacci number
int fib (int n)
{
   if (n == 0)  return 0;
   else if (n == 1) return 1;
        else  return  fib(n - 1) +  fib (n - 2);
}
```

Towers of Hanoi

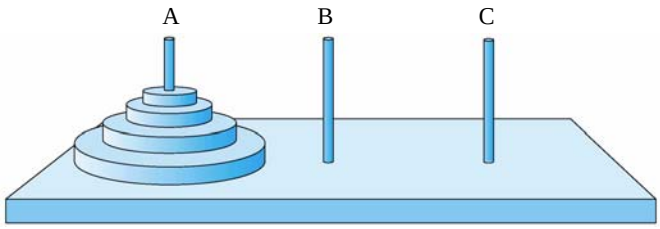
Solving the Towers of Hanoi Problem

Check out

http://www.mazeworks.com/hanoi/

- Rules of the game
- Goal - move n discs from starting peg A to the destination peg C (using an intermediary peg B)
- Move 1 disc at a time from the top of a stack, no larger disc may be placed on top of a smaller one
- Recursive solution is to break up the n disc problem into: a 1 disc problem (a stopping condition) and an n-1 disc problem

Towers of Hanoi- 4 disc example -



Tower of Hanoi Program

```
// usual stuff in here
int  main( )
{  int numDisks;        // number of disks to start with
   cout << "how  many disks do you want to play  with?" << endl;
   cin >> numDisks;
   // set the initial 3 pegs - A, B and C from left to right.
   char sourcePeg = "A";
   char destinationPeg = "C";
   char sparePeg =  "B";
   tower(sourcePeg, destinationPeg, sparePeg, numDisks);
   cout << "end of run";
   return 0;
}
// Tower function moves n disks from fromPeg to toPeg using auxPeg as an intermediary.
// It also displays a list of move instructions that transfer the disks.
void tower(char fromPeg, char toPeg, char auxPeg, int n)
{  if (n == 1)
         cout << "Move disk 1 from peg " <<  fromPeg << " to peg " <<  toPeg;
   else   // recursive block
   {      tower (fromPeg, auxPeg, toPeg, n-1);
          cout << "Move disk " <<  n << " from peg " <<  fromPeg <<  " to peg " <<  toPeg;
          tower (auxPeg, toPeg, fromPeg, n-1);
   }
} // end of Tower
```

EE 322C
Data Structures

Lecture 16

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

Good Advice ;-)

- Topics of the day: sequence containers in STL
  - All about vectors
  - More searching and sorting algorithms
  - Performance analysis using the Big O
  - Homework Problem

Terminology

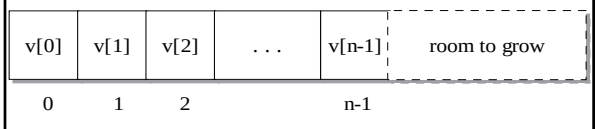
- Part of the description of an abstract data type (ADT) is a precise definition of what each operation (member function) does. There are three parts that describe its API.
  - **Operation name/action summary** - this includes an action statement that specifies the input args, the type of operation performed on elements of the data structure, and the output values
  - **Preconditions** - necessary conditions that must apply to the input args and the current state of the object for successful execution of the operation
  - **Postconditions** - changes in the data of the structure caused by performing the operation

Terminology

- These are found in the documentation header comments of each defined operation and the user documentation
- From a function implementers perspective
  - preconditions should be tested for and appropriate action taken if not valid (exception handling is the preferred method)
  - the operation should be thoroughly tested to ensure that results and postconditions are correct for all valid args

Vectors

- The vector *isa* **sequence** container that provides direct access through an index and grows/shrinks dynamically at the rear as needed.
  - ✓ memory management is automatic.
  - ✓ Supports **random access** to elements, constant time insertion and removal of elements at the end, and linear time insertion and removal of elements at the beginning or in the middle.
  - ✓ Access the elements using an index or iterator



Constructors

```
vector( );
//Create an empty vector. This is the default constructor.

vector(int n, const T& value = T( ));
//Create a vector with n elements, each having a specified
//value. If the value argument is omitted, the elements are
//filled with the default value for type T. Type T must have a
//default constructor, and the default value of type T is specified
//by the notation T( ).

vector(T *first, T *last);
//Initialize the vector using the address range first . . . last.
//The notation *first and *last is an example of pointer notation
```

Declaring Vector Objects

Syntax:

```
vector <T> vectorName (size); //T is a type name
```

Examples:

```
// vector of size 5 containing the integer values 0
  vector<int> intVector(5);
// vector of size 10; each element is the empty string
  vector<string> strVector(10);
// create a char vector of 5 elements initialized to 'x'
   vector<char> cv(5, 'x');
// vector of 100 Student objects - each element is?
  vector <Student> students (100);
```


Operations

```
T& back();
```
//Return the value of the item at the rear of the vector.
//Precondition: vector must contain at least one element.

```
const T& back( )const;
```
//Constant version of back( ).

```
bool empty( )const;
```
//Return true if the vector is empty and false otherwise.

Operations

```
T& operator[ ](int i);
```
//Allow the vector element at index i to be retrieved or modified.
//Precondition: The index, i, must be in the range $0 \leq i < n$, where n is the
// number of elements in the vector.
//Postcondition: If the operator appears on the left of an assignment
// statement, the expression on the right side modifies
// the element referenced by the index.

```
const T& operator[ ](int i) const;
```
//Constant version of the index operator.

Operations

```
void push_back(const T& value);
```
//Add a value at the rear of the vector.
//Postcondition: The vector has a new element at the rear and its size
// increases by 1.

```
void pop_back();
```
//Remove the item at the rear of the vector.
//Precondition: The vector is not empty.
//Postcondition: The vector has a new element at the rear or is empty
// and its size decreases by 1.

Operations

```
void resize((int n, const T& fill = T( ));
```
//Modify the size of the vector. If the size is increased, the
//value fill is added to the elements on the tail of the vector.
//If the size is decreased, the original values at the front
//are retained.
//Postcondition: The vector has size n.

```
int size( )const;
```
//Returns the number of elements in the vector.

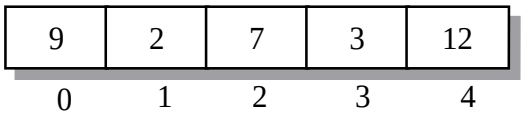
Operations

```
int capacity( )
```
//Returns the number of elements for which memory has been
//allocated. `capacity( )` is always greater than or equal to size( ).
//Memory will be reallocated automatically (by a factor of 2) if more
// than `capacity( ) - size( )` elements are inserted into the vector.

```
void reserve (int n)
```
//Preallocates memory to hold the vector elements. If n is less than
//or equal to `capacity( )`, this call has no effect. The resulting
//`capacity( )` is greater than or equal to n, but `size( )` is unchanged.
//The main reason for using `reserve( )` is efficiency and to control
//possible invalidation of iterators.

Example Use

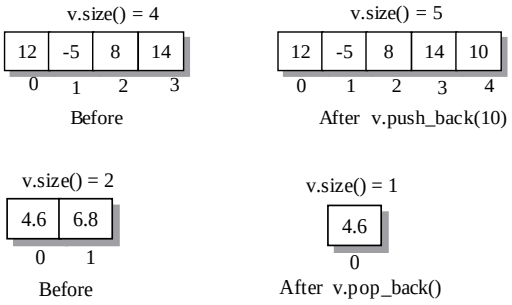
```
#include <vector>
. . .
vector<int> intVector(5);
intVector = {9,2,7,3,12};
outputVector <intVector>;
```



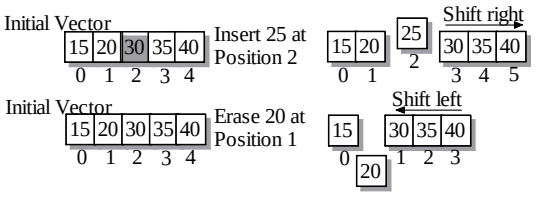
Output a Vector Example

```
// output the elements of a vector - generic
template <typename T>
void outputVector(vector<T> &v)
{ int n = v.size();
  for(int i = 0; i < n; i++)
    {cout << v[i] << "  ";}
  cout << endl;
}
```


Adding and Removing Vector Elements

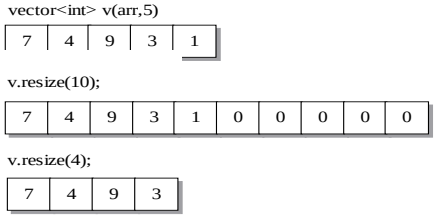


Shifting blocks of elements to insert or delete a vector item

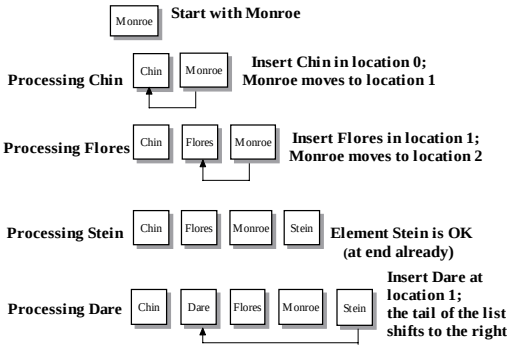


Resizing a Vector

```
int arr[5] = {7, 4, 9, 3, 1};
vector<int> v(arr,arr+5); //v initially has 5 integers
v.resize(10); //list size is doubled
v.resize(4);  //list is contracted, data is lost
```



The Insertion Sort - e.g. a vector of names



Insertion Sort Algorithm

```
// this generic template function performs the sorting
// of a vector of type T using the insertion sort
// technique

template <typename T>
void insertionSort(vector<T> &v)
{ int i, j,
  n = v.size();  // n is the size of the vector
  T temp;

  // pass through the vector in sequence examining each
  // element in turn to find and then place it into its
  // proper position in the rest of the sublist –
  // do this for v[0] ...  v[i-1], 1 <= i < n
```


Insertion Sort Algorithm

```
  for (i = 1; i < n; i++)
  {
    //index j scans down list from v[i] looking for
    //correct position to locate target. assigns it to v[j]
    j = i;
    temp = v[i];
    //locate insertion point by scanning downward as long
    //as temp < v[j-1] and we have not encountered the
    // beginning of the list
    while (j > 0 && temp < v[j-1])
      {
        //shift elements up to make room for insertion
        v[j] = v[j-1];
        j--;
      }
    // the location is found; insert temp
    v[j] = temp;
  } //end for loop
}   //end insertionsort
```


Big-O Analysis

- n is the number of elements in the vector
- the insertion sort requires n-1 passes through the vector
- For pass i the insertion occurs in the sublist v[0] to v[i-1], which requires on average i/2 comparisons
- T(n) = 1/2 + 2/2 + 3/2 + 4/2 + . . . (n-1)/2
- which is ~equal to n(n-1)/4
- Worst case is when the vector is sorted in the opposite order requiring i comparisons on each pass
- T(n) = 1 + 2 + 3 + 4 + . . . (n-1) ~= n(n-1)/2
- In both cases it is still quadratic - $O(n^2)$
- Best case is when the vector is already sorted or nearly sorted - requiring ~ O(n) comparisons

Iterators

- Special pointer objects used to cycle through the contents of a container - declared `iterator`
- Five kinds of iterators
  - Random access - store and retrieve values anywhere
  - Bidirectional - store/retrieve values in either forward or backward direction
  - Forward - store/retrieve values in forward direction only
  - Input - retrieve in forward direction
  - Output - store in forward direction
- Can increment, decrement and apply *
- For vector, e.g.
  - `vector<char> v(10); // create a vector of 10 chars`
  - `vector<char>::iterator p;`
    `// create an iterator named p for char vectors`
  - Use functions `begin( ), end( )` and `iterator` arithmetic to move through values

Homework Problem – Part I

- Dun & Bradstreet Credit Report
  - A *credit report* consists of
  - 125 sections (named 1, 2, 3, . . . , 124, 125)
  - Sections consists of text
    - The amount of text is unknown and varies with each section
    - The text may be coded, but for purposes of storage we treat each section as a string of characters
- Due Friday
  - Create the class StoreDBReport to encapsulate one report
    - Use STL classes (which two are appropriate?)
    - What is the simplest solution that solves the problem?
  - And implement the following interface operations
    - putSection
    - getSection

Homework – Part I

- Designing the Class StoreDBReport
  - What is the logical or modeling view of a report?
    - What does a report look like?
    - What do you want to do with the report?
    - What do you need to do this?
  - What does ***encapsulate*** a D&B report mean?
    - What needs to be localized?
    - How do you do that?
    - What needs to be public? Private?
    - What do you need for the public interface?
    - How do you implement the logical structure of the report?
      - Do you hide the implementation?
      - If so, how? If not, why not and what do you make public?

EE 322C
Data Structures

Lecture 17

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

Today

- Society of Hispanic Professional Engineers
  - Frightening Fiesta Fundraiser
  - Last class before Halloween: dress in a costume
  - Wearing the scariest costume I could put together ;-)
- Quiz on Monday
  - Lectures 12-17
  - 30 minutes
- Next Topics: sequence containers in STL
  - Lists - Today
  - Stacks - Monday
  - Queues – next Wednesday (Matt)

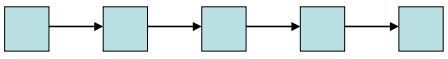
Lists

- The list *isa* sequence container that stores elements by position.
  - ✓ List containers do not permit direct access
    - No [ ] operator as in vectors
    - must start at the first position (front) and move from element to element until you locate the data value.
    - bi-directional access allowed
  - ✓ The power of a list container is its ability to efficiently add and remove items at any position in the sequence.

Lists

- List uses a list iterator
  - Is a generalized pointer that moves through a list element by element… forward or backward
  - At any point, the * operator accesses the value of a list item.
  - Accessed by using the scope operator ::

Model of a Singly Linked List Object

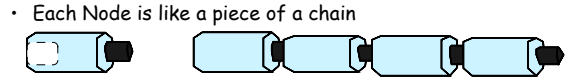


front **back**

The links are implemented as pointer variables

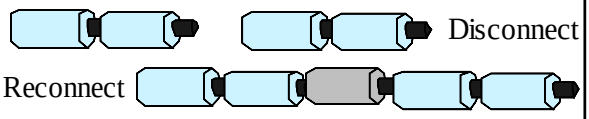
Linked List Nodes

- Each Node is like a piece of a chain



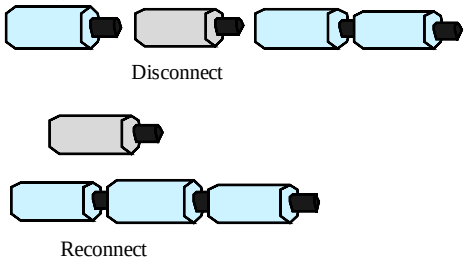
Individual Piece Pop Chain

- To insert a new link, break the chain at the desired location and simply reconnect at both ends of the new piece.



Linked List Nodes

- Removal is like Insertion in reverse.



Linked lists

Each node contains a value and a pointer to the next node in the list.

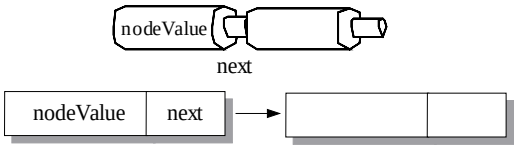
```
class node
{int nodevalue; // the data element
 node *next; // pointer to the next node
 // other stuff here
};
```

The list begins with a pointer to the first node of the list and terminates when a node has a NULL next pointer.

```
node *front = null, *temp;
```


Node Composition

- An individual Node is composed of two parts, a Data field containing the data stored by the node, and a Pointer field that contains the address of the next Node in the list.



Inserting/Deleting at the front

Insert

Set the pointer in the new node to the previous value of front. update front to point at the new node.

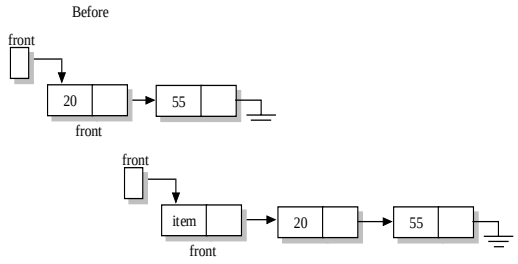
```
front = new node (value, front);
```

Erase

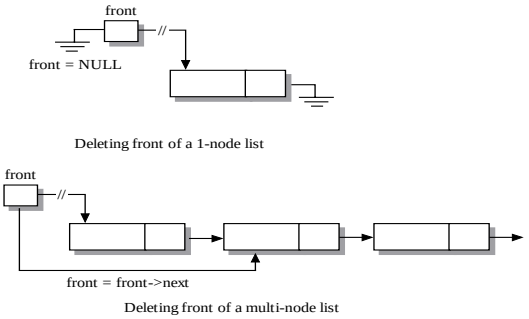
assign front the pointer value of the first node, and then delete the node.

```
temp = front;
front = front -> next;
// deallocate node at temp
```


Inserting at the Front of a Linked List



Deleting From the Front of a Linked List



Inserting/Erasing inside

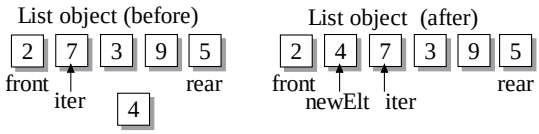
Maintain a pointer to the current list node and a pointer to the previous node.

```
node *curr,  *previous;
```

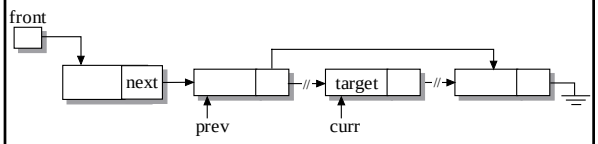
Erase: Change the pointer value in the previous node to point to the one after the node being deleted.

```
previous -> next  =  curr -> next;
```

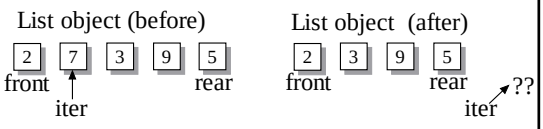

Inserting an element into a list



Removing a Target Node



Erasing an element from a list



Insert/Delete at the back

Maintain a pointer to the last list node that has value NULL when the list is empty.

```
node *back = NULL;
```

Assign a "back" pointer the address of the first node added to the list.

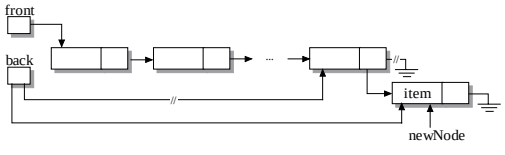
```
back = front;
```

To add other nodes at the back:

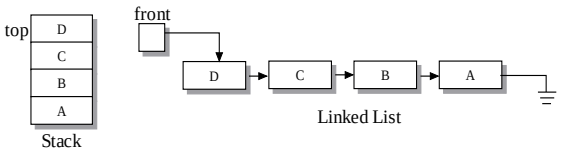
1) allocate a new node
2) assign the pointer in node "back" to point to the new node
3) assign the pointer "back" the address of the new node.

```
curr  =  new node (value, NULL);
back -> next = curr;
back = curr;
```


Inserting at the back



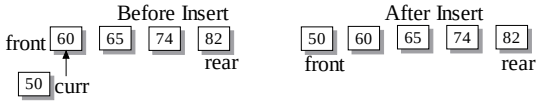
What is the Back of the List?



Maintaining an Ordered List

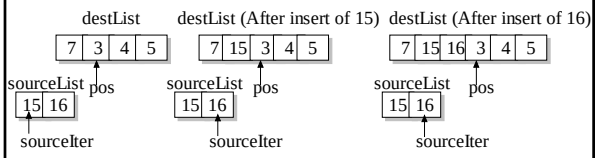
Position the iterator curr at the front of the list.

Insert 50 in the list:



Splicing two lists

sourceList.splice (destList, pos);



List Constructors

`list();`

- Create an empty list -- the default constructor

`list(int n, const T&value = T());`

- Create a list with n elements, each having a specified value.
- If the value argument is omitted, the elements are filled with the default value for type T.
- Type T must have a default constructor, and the default value of type T is specified by the notation T().

`list(T *first, T *last);`

- Initialize the list, using the address range [first, last).

List Operations

`T& back();`

- Return the value of the item at the rear of the list.
- Precondition: The list must contain at least one element.

`bool empty() const;`

- Return true if the list is empty, false otherwise

`T& front();`

- Return the value of the item at the front of the list.
- Precondition: The list must contain at least one element.

List Operations

`void push_back(const T& value);`

- Add a value at the rear of the list.
- Postcondition: The list has a new element at the rear, and its size increases by 1.

`void pop_back();`

- Remove the item at the rear of the list.
- Precondition: The list is not empty.
- Postcondition: The list has a new element at the rear or is empty, and size decreases by 1

List Operations

`void push_front(const T& value);`

- Add a value at the front of the list.
- Postcondition: The list has a new element at the front, and its size increases by 1.

`void pop_front();`

- Remove the item at the front of the list.
- Precondition: The list is not empty.
- Postcondition: The list has a new element at the front or is empty.

`int size() const;`

- Return the number of elements in the list.

List Iterator Operations

`iterator begin();`

- Returns an iterator that references the first position (`front`) of the list. If the list is empty, the iterator value `end()` is returned.

`const_iterator begin();`

- Returns a `const_iterator` that points to the first position (`front`) of a constant list.

List Iterator Operations

`iterator end();`

- Returns an iterator that signifies a location immediately past the range of actual elements.
- A program must not dereference the value of `end()` with the * operator

`const_iterator end();`

- Returns a `const_iterator` that signifies a location immediately out of the range of actual elements in a constant list.
- A program must not dereference the value of `end()` with the * operator

List Iteration Operations

`void erase(iterator pos);`

- Erase the element pointed to by pos.
- Precondition: The list is not empty.
- Postcondition: The list has one fewer element.

`void erase(iterator first, iterator last);`

- Erase all list elements within the iterator range [first, last].
- Precondition: The list is not empty.
- Postcondition: The size of the list decreases by the number of elements in the range

List Iteration Operations

`iterator insert(iterator pos, const T& value);`

- Insert value before pos, and return an iterator pointing to the position of the new value in the list.
- The operation does not affect any existing iterators.
- Postcondition: The list has a new element.

Other useful operations can be found at:

http://www.cppreference.com

List Iteration Operations

***** Accesses the value of the item currently pointed to by the iterator.
if we have list<int>::iterator iter;

```
*iter;
```

++ Moves the iterator to the next item in the list.

```
iter++;
```

-- Moves the iterator to the previous item in the list.

```
iter--;
```

== Takes two iterators as operands and returns true when they both point at the same item in the list.

```
iter1 == iter2
```

!= Returns true when the two iterators do not point at the same item in the list.

```
iter1 != iter2
```

Simple List Example

```
// example of list processing basics
#include <iostream>
#include <list>
using namespace std;
int main()
{
   list<int> lst; // create an empty list
   int i;
   for(i=0; i<10; i++) lst.push_back(i);
        // add items to the list
   cout << "Size = " << lst.size() << endl;
   cout << "Contents: ";
   list<int>::iterator p = lst.begin();
   while(p != lst.end())
   {
        cout << *p << " ";
        p++;
   }
   cout << endl << endl;
```



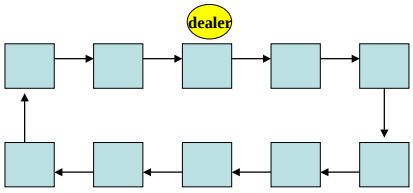
```
   // change the contents of the  list
   p = lst.begin();
   while(p != lst.end())
   {
        *p = *p + 100;  // add 100 to each item
        p++;
   }
   cout << "Contents modified: ";
   p = lst.begin();
   while(p != lst.end())
   {
        cout << *p << " ";
        p++;
   }
   cout << endl << endl;
   return 0;
}
```


Simple List Example

Or go backwards

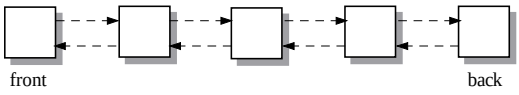
```
cout << "List printed backwards:\n";
p = lst.end();
while(p != lst.begin())
{
      p--; // decrement pointer before using
      cout << *p << " ";
}
```


Model the Poker Table as a Circular Forward-Only List



You could build this data type by specializing the list data type to logically connect the back to the front

Model of a Doubly Linked List Object

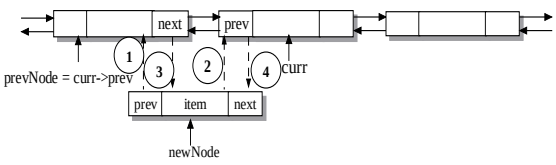


The links are implemented as pointer variables

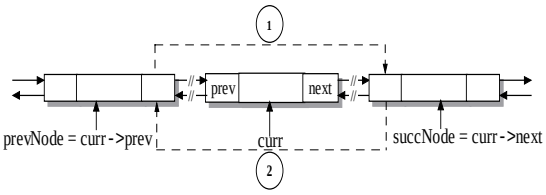
Doubly linked lists

- Provides the most flexible implementation for the sequential list.
  - Its nodes have pointers to the next and the previous node, so the program can traverse a list in either the forward or backward direction.
- Traverse a list by starting at the first node and follow the sequence of next nodes until you arrive back at the header.
- To traverse a list in reverse order, start at the last node and follow the sequence of previous nodes until arriving back at the header.

Inserting a Node at a Position

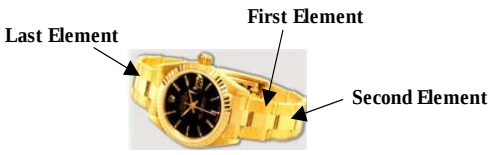


Inserting a Node at a Position



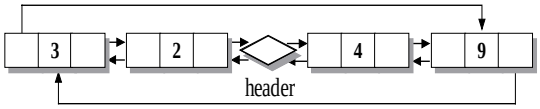
Circular Doubly Linked Lists

- A Watch Band provides a good Real Life analogue for this Data Structure



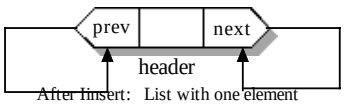
Circular Doubly Linked Lists

- Implemented on a Computer it might look something like this.

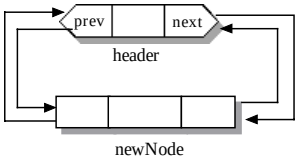


Updating a Doubly Linked List

Before Insert: Empty list



After Insert: List with one element



Hints on Debugging and Testing

- Scaffolding
  - Code used to test and debug
  - Not permanent – often thrown away
    - Print statements
    - Test drivers
- Easiest way to debug:
  - Print out important data
  - At critical points in the program
- Easiest way to test:
  - Test driver that performs various operations
    - Declares variables – perhaps with initialization
    - Call functions and class operations
    - Print input and results
  - Eg, see list example

Sample Test Driver for Homework

```
// example of homework test driver
#include . . .
#include <string>
#include <vector>
. . .
//class definition of storeDBReport
. . .
using namespace std;
int main()
{
   // first create any variables needed
   // create an empty report
   storeDBReport dbr;
   . . .
   // exercise your class code in various ways
   int i;
   for(i=0; i<125; i++) putSection(i, "default section text");
   putSection(23,  "this is section 23 – changed from the default");
   putSection(126, "this is an exception test case");
   putSection(23,  "this is the revised section 23")
   . . .
   // Print out the results of the test
   for(i=0; i<125; i++)
        cout << i << "-" <<getSection(i) << endl;
}
```

EE 322C
Data Structures

Lecture 18

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

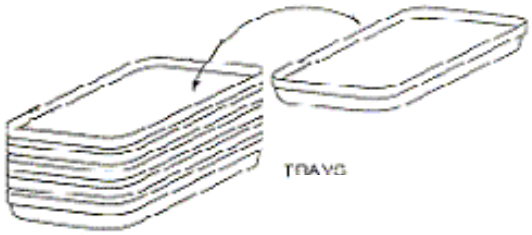
Today

- Stacks
- Homework – Part II
- Quiz

What is a Stack?

- **ADT level:** A stack is an ordered group of homogeneous items (elements), in which the removal (pop) and addition (push) of stack items can take place only at the top of the stack.
- Only the top item is usable
- A stack is a LIFO "last in, first out" structure.
- A stack is used:
  - Extensively in compilers and O/S support
  - Program run-time support
  - Implementing any concept that behaves like a stack

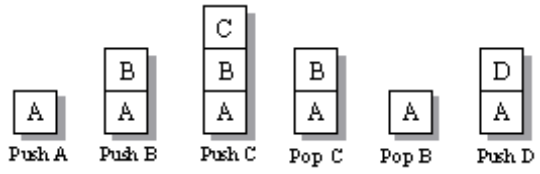
Stack of Trays



Stacks of Money

Pushing/Popping a Stack

- Because a pop removes the item last pushed onto the stack, we say that it enforces a LIFO discipline



Stack Operations

- Constructors
  - **stack**( ); //Creates an empty stack
- Comparison Operators
  - The following comparison operators are defined:
    = =, <, < =, ! =, >, > =

Stack Operations

`void push(const T& item);`

- Insert the argument item at the top of the stack.
- Postcondition: The stack has a new item at the top.

`void pop( );`

- Remove the item from the top of the stack.
- Precondition: The stack is not empty
- Postcondition: Either the stack is empty or the stack has a new topmost item from a previous push.

Stack Operations

`int size() const;`

- Return the number of items on the stack.

`T& top() const;`

- Return a reference to the value of the item at the top of the stack.
- Precondition: The stack is not empty.

`const T& top() const;`

- Constant version of top().

`bool empty(); const`

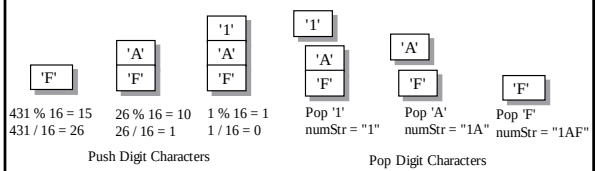
- Check whether the stack is empty. Return `true` if it is empty and `false` otherwise.

Example stack declarations

- `#include <stack>`
- `stack <int>  intstk;`
- `stack <char> digits;`
- `stack <Card> deck;`
- `stack <Student> pile;`

Using a Stack to Convert a Decimal Number to Hex

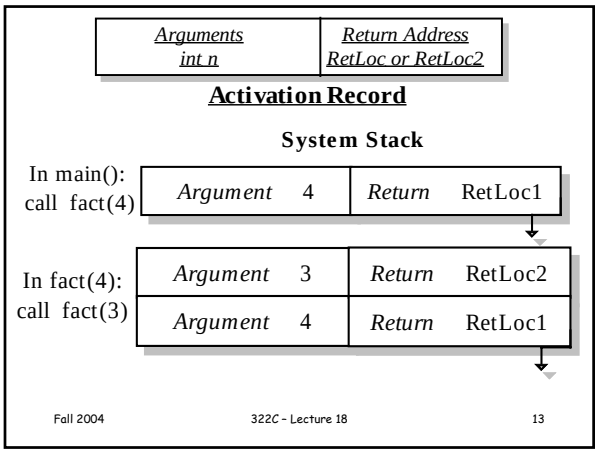
Convert 431 $_{10}$ to ??? $_{16}$



Look at example code

System Support

- Supports Recursion
- The system maintains a stack of activation records that specify:
  1. the function arguments
  2. the local variables/objects
  3. the return address
- The system pushes an activation record onto the system stack when calling a function and pops it when returning.



Used to Evaluate Expressions

Handles Postfix/RPN Expression Notation

- places the operator after its operands
- easy to evaluate using a stack to hold operands.
- The rules:
  1. Immediately push an operand onto the stack.
  2. For a binary operator, pop the stack twice, perform the operation, and push the result back onto the stack.
  3. At the end a single value remains on the stack. This is the value of the entire expression.

Used to Evaluate Expressions

Infix notation

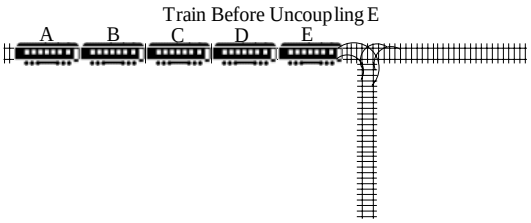
- A binary operator appears between its operands.
- More complex than postfix, because it requires the use of operator precedence and parentheses.
- Most compilers convert infix to RPN and use the above process

RPN (Reverse Polish Notation) expression 2 3 +

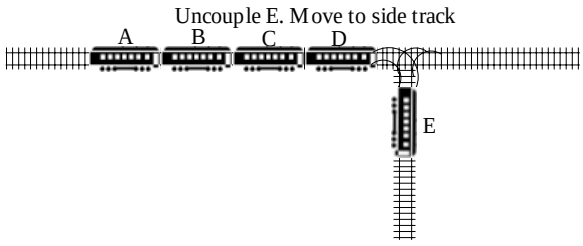
Scan of Expression and Action	Current operandStack
1. Identify 2 as an operand. Push integer 2 on the stack.	2
2. Identify 3 as an operand. Push integer 3 on the stack.	3 2
3. Identify + as an operator Begin the process of evaluating +.	3 2
4. getOperands() pops stack twice and assigns 3 to right and 2 to left.	operandStack empty
5. compute() evaluates left + right and returns the value 5. Return value is pushed on the stack.	5

Uncoupling Stack Elements

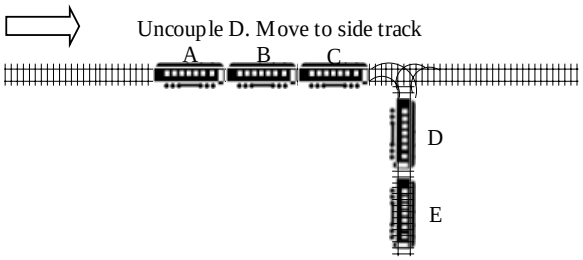
Get C out of the middle of the stack



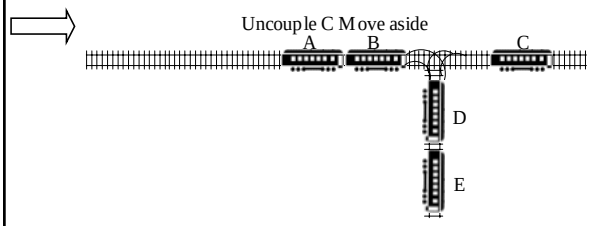
Uncoupling Stack Elements



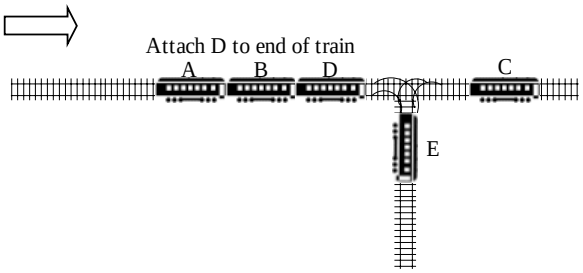
Uncoupling Stack Elements



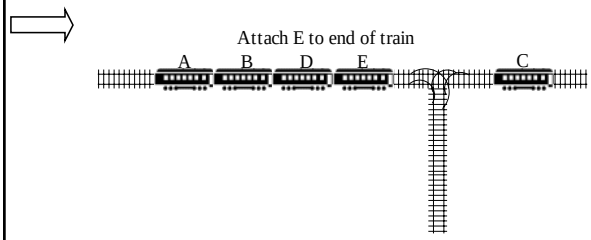
Uncoupling Stack Elements



Uncoupling Stack Elements



Uncoupling Stack Elements



Homework Part II

- In the real development from which the homework is taken (The D&B Credit Report System),
  - Space was of fixed sizes both in main memory and on disk.
  - So we had to break string up into pieces
  - For the example here we will only do that inside the report class
- Change in requirements:
  - Inside the storeDBReport class, strings can only be 128 characters in length
  - Outside the class they can still be of arbitrary size (ie, the interface to the class stays the same)

Homework Part II

- Homework:
  - Change the internal representation of report structure to satisfy this requirement (ie the private part)
  - Change getSection and putSection so that they do the appropriate thing when getting and putting sections
- Hints:
  - Use another STL data structure we have discussed to decompose and recompose the strings put and gotten from the report

EE 322C
Data Structures

Lecture 19

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

Today

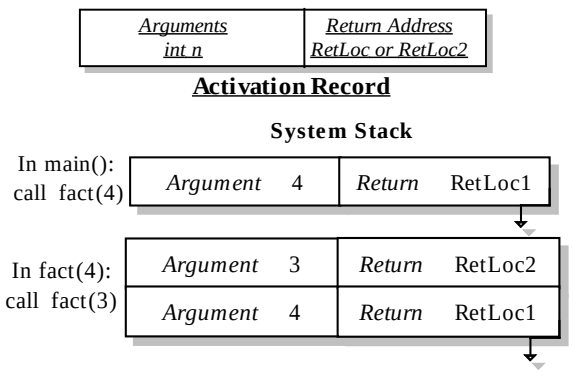
- Topics: adaptor containers in STL
  - An adaptor container uses another STL container as its implementation, e.g., a stack can be implemented using vector, list or deque (default), and a queue can be implemented using list or deque (default)
  - Stacks continued
  - Queues
    - Deque
    - Queue ADT
    - Radix Sort
    - Priority Queue

More example stack declarations

- `#include <stack>`
- `stack <int> intstk;`
- `stack <int, list <int> > intstk;`
- `stack <Card, vector <Card> > deck;`

System Stack Uses

- Program call stack (also supports Recursion)
- The system maintains a stack of activation records that specify:
  1. the function arguments
  2. the local variables/objects
  3. the return address
- The system pushes an activation record onto the system stack when calling a function and pops it when returning.
- Remember the factorial example - e.g. call to find the factorial of 4 – fact(4);



Used to Evaluate Expressions

Handles Postfix/ReversePolishNotation Expressions

- Places the operator after its operands
- Easy to evaluate using a stack to hold operands.
- The rules:
  1. Immediately push an operand onto the stack.
  2. For a binary operator, pop the stack twice, perform the operation, and push the result back onto the stack.
  3. At the end a single value remains on the stack. This is the value of the entire expression.

Used to Evaluate Expressions

Infix notation
- A binary operator appears between its operands.
- More complex than postfix, because it requires the use of operator precedence and parentheses.
- Most compilers convert infix to RPN and use the above process.

RPN (Reverse Polish Notation) expression 2 3 +

Scan of Expression and Action	Current operandStack
1. Identify 2 as an operand. Push integer 2 on the stack.	2
2. Identify 3 as an operand. Push integer 3 on the stack.	3 2
3. Identify + as an operator Begin the process of evaluating +.	3 2
4. getOperands() pops stack twice and assigns 3 to right and 2 to left.	operandStack empty
5. compute() evaluates left + right and returns the value 5. Return value is pushed on the stack.	5

Double Ended Queue (deque)

A deque is a combination LIFO and FIFO Data Structure.

Elements are inserted in the Front or Back of the deque and are removed from the Front or Back.

Deque Constructors

`deque( );`
- Create an empty deque.

`deque(int num, const T &val = T( ) );`
- Create a deque with num elements of the value val.

`deque(const T, &ob);`
- Create a deque with the same elements as ob.

`deque(InIter start, InIter end);`
- Create a deque with the elements in the range specified by iterator values of start and end.

Deque Operations

Comparison operators: = =, <, < =, ! =, >, > =

`bool empty() const;`
- Check whether the deque is empty. Return true if it is empty and false otherwise.

`T& front();`
- Return a reference to the value of the item at the font of the deque without removing the item.
- Precondition: The deque is not empty.

`const T& front() const;`
- Constant version of front().

`T& back();`
- Return a reference to the value of the item at the back of the deque without removing the item.
- Precondition: The deque is not empty.

`const T& back() const;`
- Constant version of back().

Deque Operations

`void push_back (const T& item);`
- Insert the argument item at the back of the deque.
- Postcondition: The deque has a new item at the back

`void push_front (const T& item);`
- Insert the argument item at the front of the deque.
- Postcondition: The deque has a new item at the front.

`void pop_front ();`
- Remove the item from the front of the deque.
- Precondition: The deque is not empty.
- Postcondition: A new element at the front of the deque is revealed or the deque is empty.

Deque Operations

`void pop_back ();`

- Remove the item from the back of the deque.
- Precondition: The deque is not empty.
- Postcondition: A new element at the back of the deque is revealed or the deque is now empty.

`int size( ) const;`

- Return the number of elements in the deque.

Iterator operations: just like `list`

`erase` and `insert`: just like `list`

Other useful operations: see cppreference.com

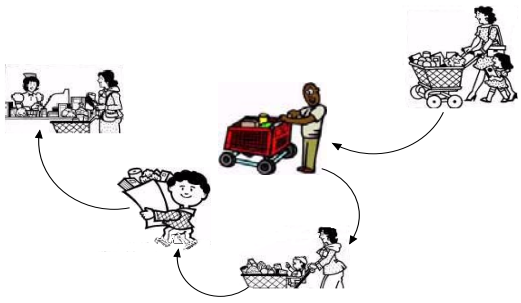
Double Ended Queue (deque)

- Is used as an implementation class for many other STL classes because of its flexibility
- Every class in the STL has an allocator defined for it. The allocator is an object that manages the memory allocation scheme for that container.
  - We will use the default allocators for each class, but you could define your own specialized allocator if you wanted to

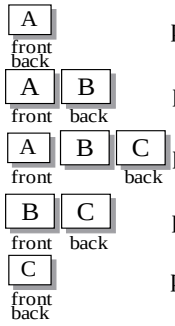
Queue

- A queue is a sequence container that supports a first-in-first-out (FIFO) data discipline. Sometimes called first-come-first-served.
  - Insertion operations (`push( )`) occur at the back of the sequence
  - deletion operations (`pop( )`) occur at the front of the sequence.
- There are lots of examples of queues in the real world, as well as in the system software world

Grocery Store Checkout: An Example of a Queue



The Queue



push A

push B

push C

pop A

pop B

A Queue is a FIFO (First in First Out) Data Structure. Elements are inserted in the Rear of the queue and are removed at the Front.

Queue Operations

- Constructor:
  - `queue();`
    - Create an empty queue.
- Basic Operations
  - `bool empty() const;`
    - Check whether the queue is empty. Return true if it is empty and false otherwise.
  - `T& front();`
    - Return a reference to the value of the item at the font of the queue without removing the item.
    - Precondition: The queue is not empty.
  - `const T& front() const;`
    - Constant version of front().

Queue Operations

`void push(const T& item);`

- Insert the argument item at the back of the queue.
- Postcondition: The queue has a new item at the back.

`void pop();`

- Remove the item from the front of the queue.
- Precondition: The queue is not empty.
- Postcondition: The element at the front of the queue is the element that was added immediately after the element just popped or the queue is empty.

`int size() const;`

- Return the number of elements in the queue.

Queue Scheduler Example

```
/* the program creates and outputs the interview schedule for a
   personnel director.  The schedule is constructed using a queue
   of appointment times by reading the times from the keyboard.
   By then cycling through the queue, the appointment times are
   output.
*/
#include <iostream>
#include <queue>
using namespace std;
int main()
{
   int interviewTime;  // using a simple 24 hour clock
   // queue to hold hourly appointment time for job applicants
   queue <int> apptQ;
   // build the schedule from inputs
   cout << "First interview of the day: ";
   cin >> interviewTime;
```


Queue Scheduler Example

```
   // construct the queue until input is 17:00 or later
   while (interviewTime < 17)
   {
        //  push the interview time on the queue
        apptQ.push(interviewTime);
        // prompt for the next interview time
        cout << "Next interview: ";
        cin >> interviewTime;
   }
   // output the day's appointment schedlule
   cout << endl << "Appointment Schedule" << endl;
   while (!apptQ.empty())
   {
        interviewTime = apptQ.front();
        // pop the next applicant appointment time and output it
        apptQ.pop();
        cout << "   " << interviewTime << endl;
   }
   return 0;
}
```


Sorting with Queues

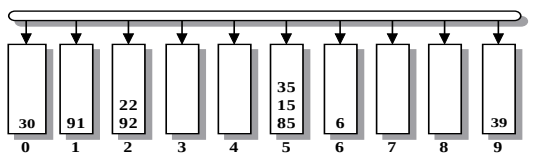
The radix (bin) sort algorithm

- Orders an integer vector by using queues (bins).
- This sorting technique has running time O(n) but has only specialized applications.
- The more general in-place O(n log2n) sorting algorithms are preferable in most cases.

The Radix Sort

Order a sequence of 2-digit numbers in 10 bins from smallest number to largest number.

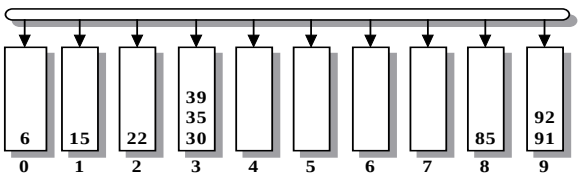
Initial Sequence: 91 6 85 15 92 35 30 22 39
Pass 0: Distribute the numbers into bins according to the 1's digit (10^0).



New Sequence: 30 91 92 22 85 15 35 6 39

The Radix Sort

Pass 1: Take the new sequence and distribute the numbers into bins determined by the 10's digit (10^1). Then un-queue them L to R, front to back



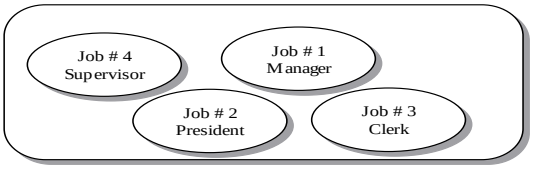
Final Sequence: 6 15 22 30 35 39 85 91 92

Priority queue

- `Pop()` returns the highest priority item (assumed to be the largest value).
- The `push()` and `pop()` operations have running time $O(\log_2 n)$
- Normally implemented by a heap data structure

Priority Queue

A Special form of queue from which items are removed according to their designated priority and not the order in which they entered.



Items entered the queue in sequential order but will be removed in the order #2, #1, #4, #3. As an HR person might process the resume queue.

Priority Queue Operations

- Constructor
  - `priority_queue();`
    - Create an empty `priority queue`. Type T must implement the operator <.
- Basic Operations
  - `bool empty() const;`
    - Check whether the `priority queue` is empty. Return `true` if it is empty, and `false` otherwise.
  - `void pop();`
    - Remove the item of highest priority from the queue.
    - Precondition: The `priority queue` is not empty.
    - Postcondition: The `priority queue` has 1 less element or is empty.

Priority Queue Operations

- `void push(const T& item);`
  - Insert the argument item into the `priority queue`.
  - Postcondition: The `priority queue` contains a new element.
- `int size() const;`
  - Return the number of items in the `priority queue`.
- `T& top();`
  - Return a reference to the item having the highest priority without removing the item.
  - Precondition: The `priority queue` is not empty.
- `const T& top();`
  - Constant version of `top()`.

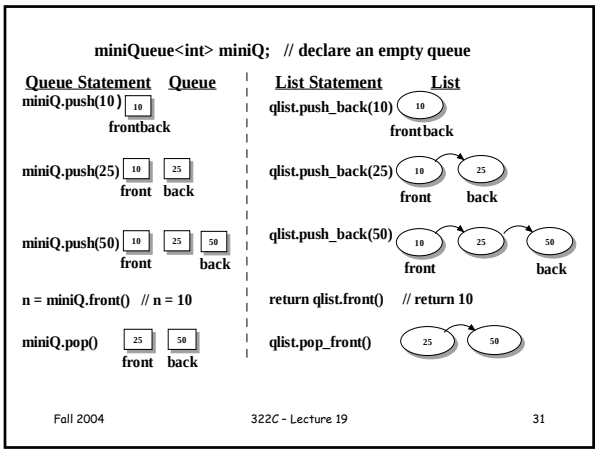
Example

```
priority_queue <int> mypq;
int n;
mypq.push (20);
mypq.push (10);
mypq.push (67);
n = mypq.pop ( );
cout << n;
```

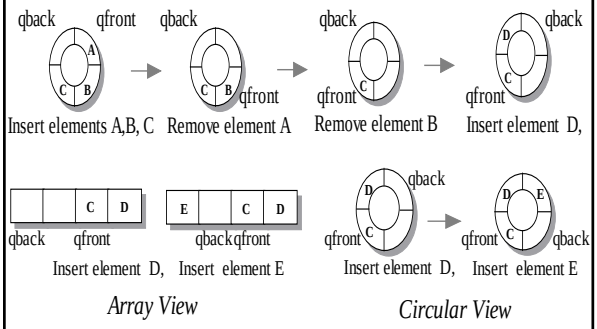

miniQueue

The miniQueue class

- Provides a class with STL queue class interface.
- Uses the list class by object composition.



The Bounded queue

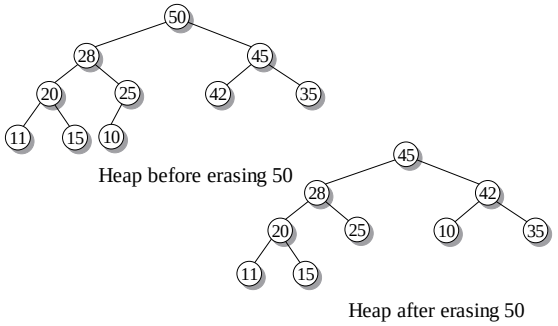


Implementation Considerations

Implementing a queue with a fixed-size array

- Indices `qfront` and `qback` move circularly around the array.
- Gives O(1) time `push()` and `pop()` operations with no wasted space in the array.

Removing Elements From a Heap



EE 322C
Data Structures

Lecture 19 – Quiz Results

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

General Problems

- True & False
  - Almost every one missed 1 and 7
  - In 1 the key word is "ordered" – that makes it false
  - In 7 the key word is "free" which in this case means "arbitrary"
- Multiple choice
  - Many missed 9.4 – an array does allows efficient insertion and deletion of back elements
  - In 12, vectors do not provide efficient arbitrary capacity – at some point a new structure must be allocated and copied into
  - In 13, lists do not have direct access

General Problems

- Completion
  - 14: preconditions and postconditions
  - 15: nested (for) loops
  - 16: iterators
  - 17: stacks and queues
  - 18: performance, efficiency or complexity
  - 19: doubly linked lists

Overall Distribution

- 46.5 x
- 46 xxxx
- 45.5 xxx
- 44.5 x
- 43.5 x
- 43 xxx
- 42.5 xx
- 42 xx
- 41.5 xxxx
- 41 xxxx
- 40.5 x
- 40 xxx
- 39.5 xx
- 39 x
- 38.5 xxx
- 38 x
- 37.5 x
- 37 xx
- 36.5 xxx
- 36 x
- 35.5 xx
- 34.5 xxx
- 34 x
- 33.5 xx
- 32.5 x
- 32 x
- 31 x
- 30.5 x
- 29.5 x
- 26 x

True & False

- 14 xxx
- 12 xxxxx xxxxx xxxxx xxx
- 10 xxxxx xxxxx xxxxx xxxxx
- 8 xxxxx xxxxx xxx
- 6 xx

Multiple Choice

- 17.5 x
- 17 x
- 16.5 xxxx
- 16 xxxxx
- 15.5 xxxxx
- 15 xxxx
- 14.5 xxxx
- 14 xxxxx xxxxx
- 13.5 xxxxx xxxx
- 13 xxx
- 12.5 xxxxx
- 11.5 x
- 11 x
- 10.5 x
- 10 x

Completion

- 18 xxxxx xxxxx xxxxx xx
- 16.5 xxxx
- 15.5 x
- 15 xxxxx xxxxx xxx
- 14 xxx
- 13.5 xxxxx
- 13 xx
- 12 xxxx
- 11.5 x
- 10.5 x
- 9 xxx
- 7.5 xx
- 7 x

Grading

- Medians and Means
  - T/F – median: 10, mean: 10.28
  - MC - median: 14, mean: 14.14
  - C - median: 15, mean: 14.68
  - All - median: 40, mean: 39.11
- Uncurved:
  - 90+% 8
  - 80+% 21
  - 70+% 16
  - 60+% 10
  - <60% 2

Grading

- Curved
  - A 42.5 – 46.5 15
  - B 38 - 42 21
  - C 29.5 – 37.5 20
  - D <29.5 1

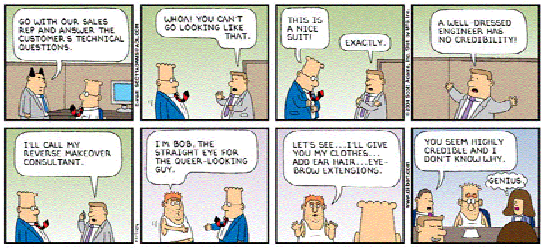
EE 322C
Data Structures

Lecture 20

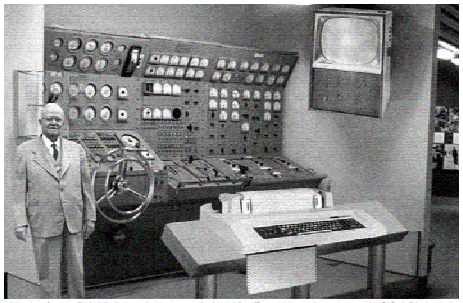
Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

All Toooo True



Popular Science 1954



Scientists from the RAND Corporation have created this model to illustrate how a "home computer" could look like in the year 1984. However the needed technology will not be economically feasible for the average home. Also the scientists readily admit that the computer will require not yet invented technology to actually work, but in years from now scientific progress is expected to solve these problems. With teletype interface and the Fortran language, the computer will be easy to use.

Announcements

- Today's topics - associative containers (elements are stored by key and not by position)
  - sets, multisets
  - maps, multimaps
- Homework – Part III
- Quiz 2 -

ADT Set Definitions

Set: is an unordered collection of ***distinct*** elements (or values) chosen from the possible values of a base data type

- **Base type:** The data type of the elements in the set
- **Cardinality:** The number of elements in a set
- **Universal set**: the set containing all values of the base data type
- **Empty set:** a set with no elements

Subset: A set X is a subset of set Y if each element in X is also in Y; if there is at least one element of Y that is not in X, then X is a proper subset of Y.

Common set binary operations:

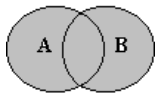
- **Union of two sets:** A set made up of all the items in either of two given sets
- **Intersection of two sets:** A set made up of all the items in both sets
- **Difference of two sets:** A set made up of all the items in the first set that are not in the second set

Implications

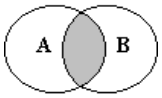
- Sets can not contain duplicates. Storing an item that is already in the set does not change the set.
- If an item is not in a set, deleting that item from the set does not change the set.
- Sets are not ordered.
- A ***multiset*** (the STL container) is like a set, except a value can occur more than once

Set Examples

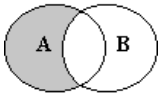
- Let's assume that we have 2 sets of integers called A and B
- A = { 1, 3, 8, 9, 10 }
- B = { 9, 6, 3, 2}
- Or as they might be declared in C++
  - set <int> A = {1,3,8,9,10};
  - set <int> B = {9,6,3,2};



Set-Union Operator + (A + B):
The set of all elements x such that x is an element in set A OR x is an element in set B.
Example: A + B = { 1, 2, 3, 6, 8, 9, 10}



Set-Intersection Operator * (A * B):
The set of all elements x such that x is an element in set A and x is an element in set B.
Example: A * B = { 3, 9}



Set-Difference Operator - (A - B):
The set of all elements x such that x is an element in set A but x is not an element in set B.
Example: A - B = { 1, 8, 10}

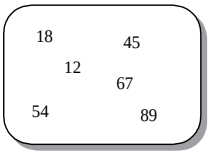
Implementing the Set ADT

Implementation as a bit vector
- Each item in the base type has a representation in each instance of a set.
- The representation is either true (item is in the set) or false (item is not in the set).
- Space is proportional to the cardinality of the base type.
- Algorithms use Boolean operations.

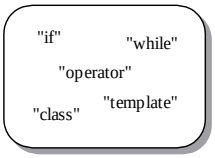
Implementation as a list/vector
- The items in an instance of a set are on a list that represents the set.
- Those items that are not on the list are not in the set.
- Space is proportional to the cardinality of the set instance.
- Algorithms use ADT List operations.

Using the STL set container
- Uses an iterator that defines an ordering of the keys
- Underlying structure is actually a bstree stored as a vector

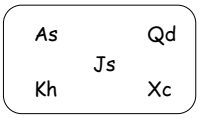
Example Sets



intSet: Set of ints

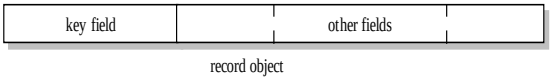


keyword: Set of strings



Hand: set of Card objects

Sets defined by a key along with other data



record object

Set Constructors/Operations

Constructors

`set();`
- Create an empty set. This is the Default Constructor.

`set(T *first, T *last);`
- Initialize the set by using the address range [first, last).

Operations

`bool set() const;`
- Is the set empty?

`int size() const;`
- Return the number of elements in the set.

Set Operations

`int count(const T& key) const;`

- Search for key in the set and return `1` if it is in the set and `0` otherwise.

`iterator find(const T& key);`

- Search for key in the set and return an `iterator` pointing at it, or `end()` if it is not found.
- *Note:* Iterators define an ordering of the keys, not positions

`Const_iterator find(const T& key) const;`

- Constant version of `find`.

Set Operations

`pair<iterator, bool> insert(const T& key);`

- If key is not in the set, insert it and then return a pair whose first element is an iterator pointing to the new element and whose second element is true. Otherwise, return a pair whose first element is an iterator pointing at the existing element and whose second element is false.
- Postcondition: The set size increases by `1` if key is not in the set.

`int erase(const T& key);`

- If key is in the set, erase it and return `1`; otherwise, return `0`.
- Postcondition: The set size decreases by `1` if key is the set.

Set Operations

`void erase(iterator pos);`

- Erase the item pointed to by pos.
- Preconditions: The set is not empty, and `pos` points to a valid set element.
- Postcondition: The set size decreases by 1.

`void erase(iterator first, iterator last);`

- Erase the elements in the range [first, last).
- Precondition: The set is not empty.
- Postcondition: The set size decreases by the number of elements in the range.

Set Iterators

`iterator begin();`

- Return an iterator pointing at the first member in the set.
- (smallest by default)

`const_iterator begin(const);`

- Constant version of `begin()`

`iterator end();`

- Return an `iterator` pointing just past the last member in the set

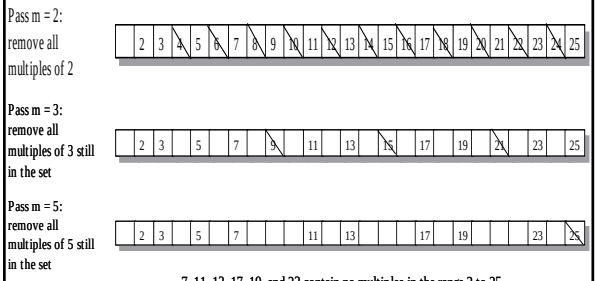
`const_iterator end() const;`

- Constant version of `end()`

Sieve of Eratosthenes

- Goal: Find all the prime numbers in the range from 2 to n
- Look at an example
- Look at the code using sets

Sieve of Eratosthenes



7, 11, 13, 17, 19, and 23 contain no multiples in the range 2 to 25

Primes {2, 3, 5, 7, 11, 13, 17, 19, 23}

STL Template Algorithms

- All template functions in the STL that are associated with many container types
- Operate on containers using ***iterators***
- There are approximately 60 of these
- For sets we have:
  - set_difference
  - set_intersection
  - set_union
  - set_symmetric_difference

Multiset Operations

- Multiset is just like a set, except a key can occur more than once.
- `int count(const T& item) const;`
  - Return the number of duplicate occurrences of item in the multiset.
- `pair<iterator, iterator> equal_range(const T& item);`
  - Return a pair of iterators such that all occurrences of item are in the iterator range[first member of pair, second member of pair).
- `iterator insert(const T& item);`
  - Insert item into the multiset and return an iterator pointing at the new element.
  - Postcondition: The item is added to the multiset.
- `int erase(const T& item);`
  - Erase all occurrences of item from the multiset and return the number of items erased.
  - Postcondition: The size of the multiset is reduced by the number of occurrences of item in the multiset.

Map

- A map is a collection of key-value pairs that associate a key with a value.
  - In a map, there is only one value associated with a key.
- A map is often called an associative array because applying the index operator with the key as its argument accesses the associated value
- The map STL class has all the same operations found in set, however the elements are pairs not a single data item
- Balanced binary search tree is used for the STL implementation

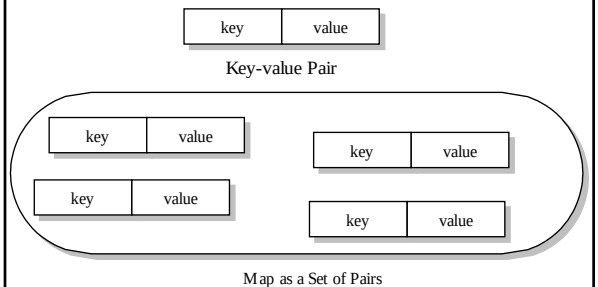
Key-Value Data

A map stores data as a key-value pair. In a pair, the *first* component is the key; the *second* is the value. Each component may have a different data type.

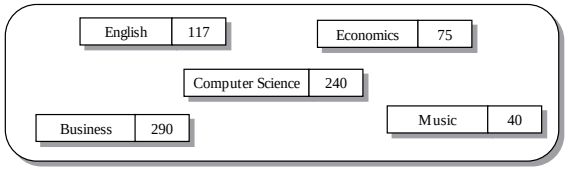
A lookup operation takes the key and returns an iterator that points to a matching (key, value) pair.

Only unique keys are allowed in maps

Maps



Map Example



degreeMajor: Map of string-int pairs

```
map <string, int> degreeMajor;
// assume values have been filled in as above
cout << degreeMajor [ "English" ];
degreeMajor [ "ECE" ] = 322;
```

Homework Part III

- History
  - Once the D&B Credit Reporting Systems was completed, it was subjected to load tests.
  - Result: only could store half the projected number of reports
  - Turned out the requirements were incorrect:
    - *A report consists of **at most** 125 sections*
    - *The number of sections depends on the report type*
    - *Half of the reports only have 5 sections*
  - Created a data structure keyed by the report type to tell me how many elements to have in the section list
  - Changed the report representation
    - from an array of pointers to the sections (strings) indexed by the section number
    - to an data structure with elements of <section identifier, pointer to section> the size of which is determined by the report type;
    - further I had to allow for the possibility that there might be more than the usual number of sections

Homework – Part III

- There are 5 report types
  - RT1 has 25 sections typically (but may more or less)
  - RT2 has 5 sections typically (but may more or less)
  - RT3 has 10 sections typically (but may more or less)
  - RT4 has 5 sections typically (but may more or less), and
  - RT5 has 15 sections typically (but may more or less)
- Assignment:
  - Define a type to capture the concept of report types
  - Define an appropriate data structure to represent the report type information (an appropriate STL we have talked about)
  - Add createReport that uses report type as a parameter and information above to create a data structure that has space for the usual number of sections for that report type
  - Change putSection and getSection to reflect this new representation.
  - Make sure that you can handle more then the usual number of sections

EE 322C
Data Structures

Lecture 21

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

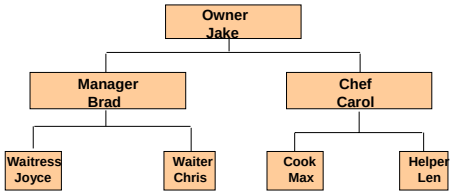
Today

- Topics of the day - start non sequential data structures:
  - Trees
  - Binary trees

Tree Definitions

- Hierarchical structures that place elements in nodes along branches that originate from a root.
- Nodes in a tree are subdivided into levels in which the topmost level holds the root node.
  - Any node in a tree may have multiple successors at the next level. Hence a tree is a non-linear structure.
- Tree terminology with which you should be familiar:
  - root | parent | child | descendents | leaf node | interior node | subtree | level | ancestors |.

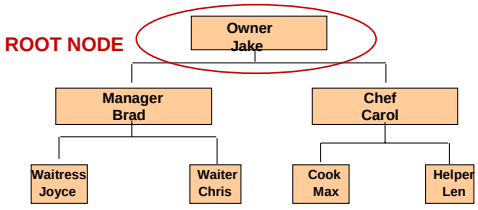
Jake's Pizza Shop



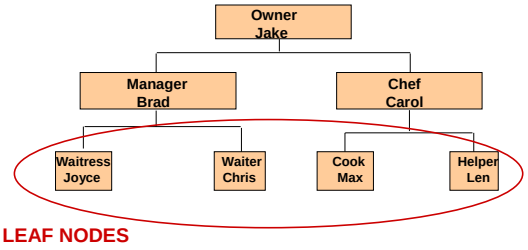
- position
- person

are the attributes of interest in an organization chart

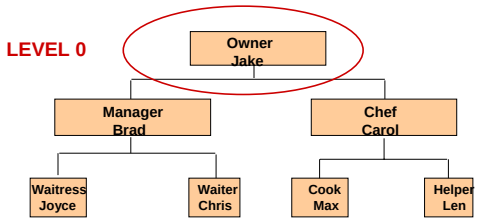
A Tree Has a Root Node



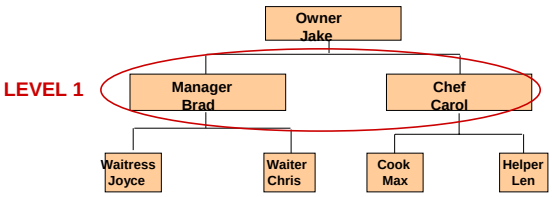
Leaf Nodes have No Children



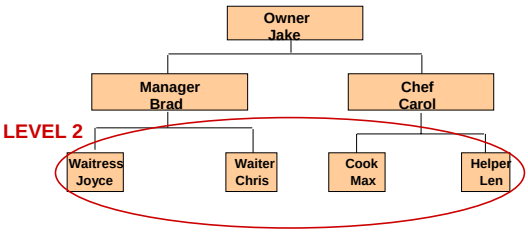
A Tree Has Levels



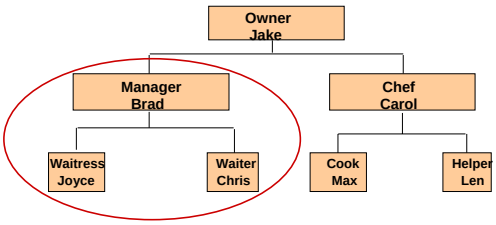
Level One



Level Two

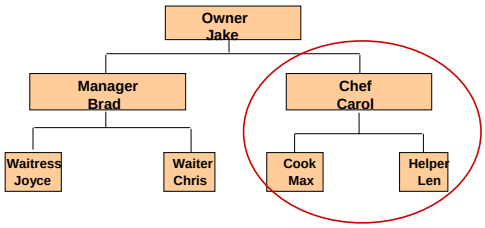


A Subtree



LEFT SUBTREE OF ROOT NODE

Another Subtree



RIGHT SUBTREE OF ROOT NODE

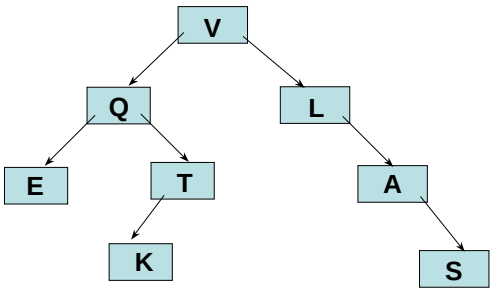
Binary Tree

A binary tree *isa* tree structure in which:

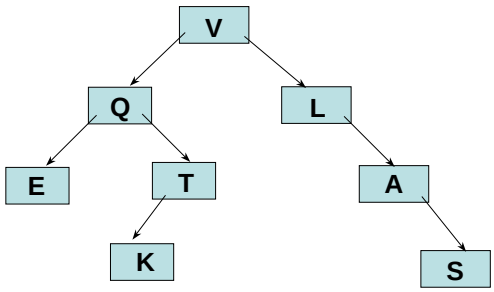
Each node can have at most two children, and in which a unique path exists from the root to every other node.

The two children of a node are called the left child and the right child, if they exist.

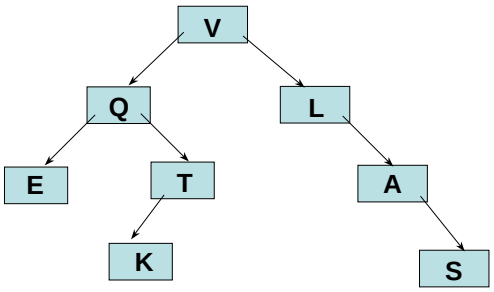
A Binary Tree



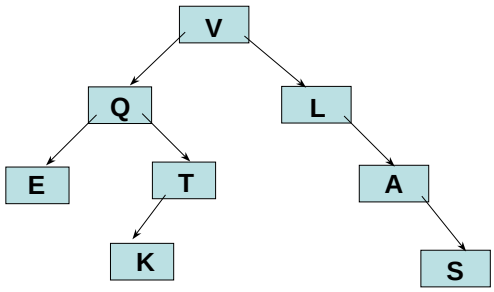
How many leaf nodes?



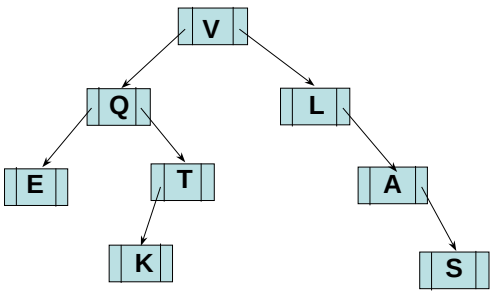
How many descendants of Q?



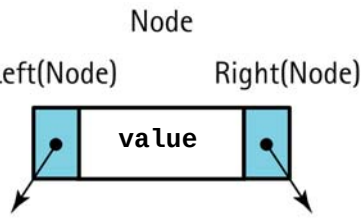
How many ancestors of K?



Implementing a Binary Tree with Pointers and Dynamic Data



Node Terminology for a BTree Node



Model Node of a Binary Tree

```
// this generic class models a node of a binary tree
template <typename T> class btnode
{ public:
    T nodeValue;  // the data value being stored
    btnode <T> *left;    // pointer to left subtree
    btnode <T> *right;   // pointer to right subtree
    btnode ( )  { nodeValue = T ( );
       left =  NULL;  right =  NULL; }
    btnode (const T &val)  {nodeValue = val;
       left = NULL;  right  = NULL; }
    btnode (const T &val,  btnode <T>  *leftTemp =  NULL,
            btnode <T> *rightTemp =  NULL)
       { nodeValue = val; left = leftTemp;
            right = rightTemp;  }
    ~btnode () { }
}
```


Model Node of a N-ary Tree

```
// this generic class models a node of a N-ary tree
template <typename T> class ntnode
{ public:
    T nodeValue;  // the data value being stored




}
```


A Binary Search Tree (BST) is . . .

A special kind of binary tree in which:

1. Each node contains a distinct data value,
2. The key values in the tree can be compared using "greater than" and "less than", and
3. The key value of each node in the tree is less than every key value in its right subtree, and greater than every key value in its left subtree.

Shape of a binary search tree . . .

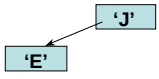
Depends on its key values and their order of insertion.

Insert the elements 'J' 'E' 'F' 'T' 'A' in that order.

The first value to be inserted is put into the root node.

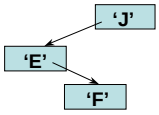
Inserting 'E' into the BST

Thereafter, each value to be inserted begins by comparing itself to the value in the root node, moving left it is less, or moving right if it is greater. This continues at each level until it can be inserted as a new leaf.



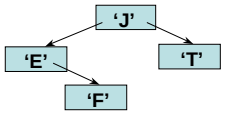
Inserting 'F' into the BST

Begin by comparing 'F' to the value in the root node, moving left it is less, or moving right if it is greater. This continues until it can be inserted as a leaf.



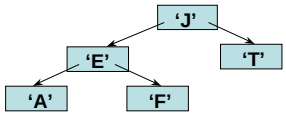
Inserting 'T' into the BST

Begin by comparing 'T' to the value in the root node, moving left it is less, or moving right if it is greater. This continues until it can be inserted as a leaf.



Inserting 'A' into the BST

Begin by comparing 'A' to the value in the root node, moving left it is less, or moving right if it is greater. This continues until it can be inserted as a leaf.

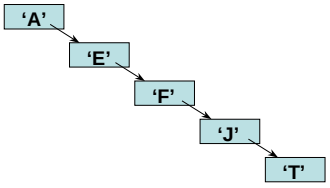


What binary search tree . . .

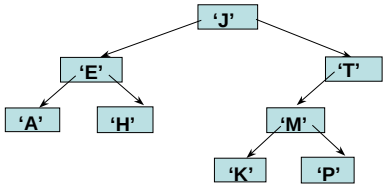
is obtained by inserting the elements 'A' 'E' 'F' 'J' 'T' in that order?

Binary search tree . . .

obtained by inserting the elements 'A' 'E' 'F' 'J' 'T' in that order.



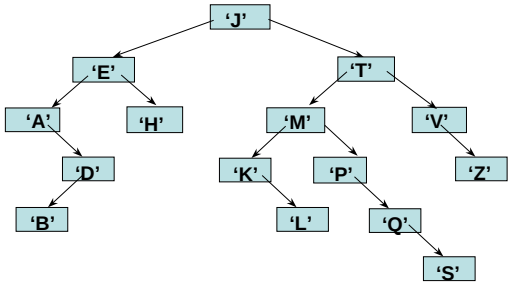
Another binary search tree



Add nodes containing these values in this order:

'D' 'B' 'L' 'Q' 'S' 'V' 'Z'

Is 'F' in the binary search tree?



Traversing Through a Tree

There are six simple recursive algorithms for tree traversal.

The most commonly used ones are:

1. inorder (LNR)
2. postorder (LRN)
3. preorder (NLR).

Another technique is to move left to right from level to level.

- This algorithm is iterative, and its implementation involves using a queue.

Recursive Btree Example

```
// algorithm for counting the # of nodes
// in a given binary tree

if tree is NULL
  return 0
else
  return CountNodes(Left(tree)) +
      CountNodes(Right(tree)) + 1
```


Recursive B-Tree Example

```
{ // in main
  int number = CountNodes(root);
}

int CountNodes(btnode *tree)
// Recursive function that counts the # of nodes
{
  if (tree == NULL)
    return 0;
  else
    return CountNodes(tree->left) +
      CountNodes(tree->right) + 1;
}
```

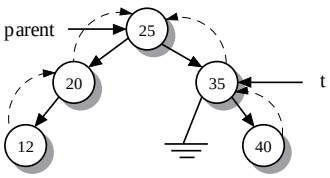

Update Operations

- Insertion is easy but deletion is hard
- The removal of an item from a binary search tree is more difficult and involves finding a replacement node among the remaining values.

Insert Operations: 1st of 3 steps

1)- The function begins at the root node and compares item 32 with the root value 25. Since 32 > 25, we traverse the right subtree and look at node 35.

Goal:
Insert the value 32

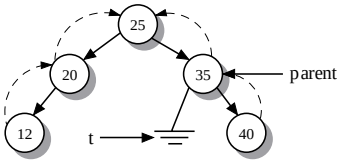


(a)

Step 1: Compare 32 and 25.
Traverse the right subtree.

Insert Operations: 2nd of 3 steps

2)- Considering 35 to be the root of its own subtree, we compare item 32 with 35 and traverse the left subtree of 35.



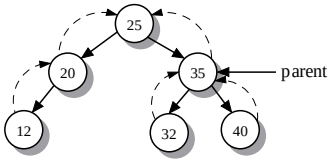
(b)

Step 2: Compare 32 and 35.
Traverse the left subtree.

Insert Operations: 3rd of 3 steps

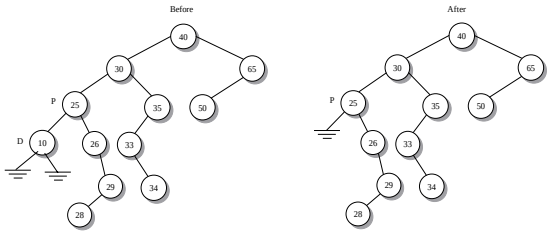
3)-Create a leaf node with data value 32. Insert the new node as the left child of node 35.

```
newNode = getSTNode(item,NULL,NULL,parent);
parent->left = newNode;
```



(c)
Step 3: Insert 32 as left child of parent 35

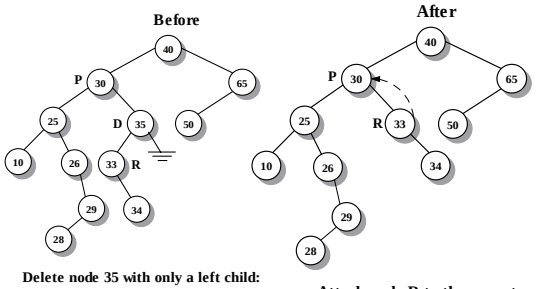
Remove - Case1: No Subtrees



Delete leaf node 10. pNodePtr->left is dNode

No replacement is necessary. pNodePtr->left is NULL

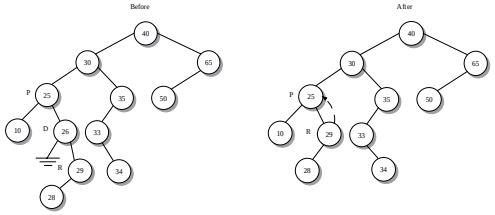
Remove - Case 2: Left Subtree



Delete node 35 with only a left child: Node R is the left child.

Attach node R to the parent.

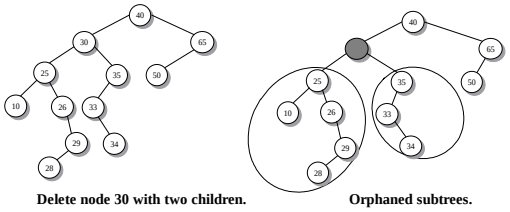
Remove - Case 3: Right Subtree



Delete node 26 with only a right child: Node R is the right child.

Attach node R to the parent.

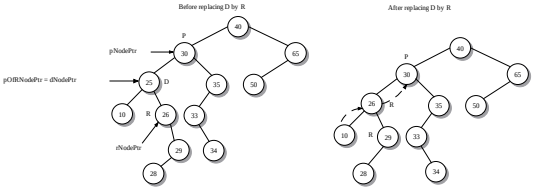
Remove - Case 4: Left/Right Subtrees



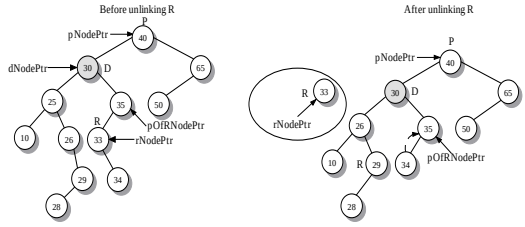
Delete node 30 with two children.

Orphaned subtrees.

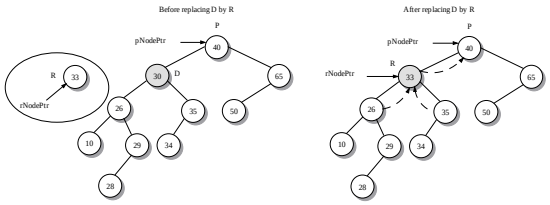
Remove – Case 4



Remove – Case 4



Removing an Item



Binary Trees

- Most effective as a storage structure if it has high density
  - data are located on relatively short paths from the root.
  - A complete, full binary tree has the highest possible density
- an n-node complete binary tree has depth int(log2n).
- At the other extreme, a degenerate binary tree is equivalent to a linked list and exhibits O(n) access times.

EE 322C
Data Structures

Lecture 22

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

Lecture 25 Announcements

- Course survey
- Finish trees
- Matrices

Survey Information

- Instructor: Dewayne Perry
- Course #: EE322C
- Unique #: 15515
- Semester: Fall 2004

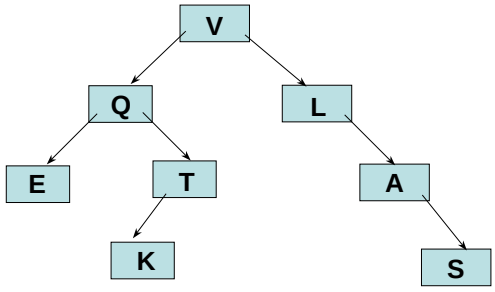
Binary Tree

A binary tree *isa* tree structure in which:

Each node can have at most two children, and in which a unique path exists from the root to every other node.

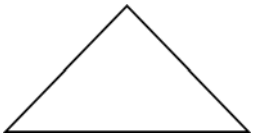
The two children of a node are called the left child and the right child, if they exist.

A Binary Tree



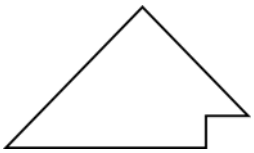
Definitions

- **Full Binary Tree:** A binary tree in which all of the leaves are on the same level and every nonleaf node has two children

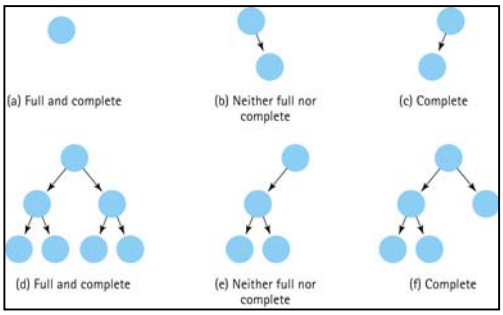


Definitions (cont.)

- **Complete Binary Tree:** A binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far to the left as possible



Different Types of Binary Trees



Traversing Through a Tree

There are six simple recursive algorithms for tree traversal.

The most commonly used ones are:

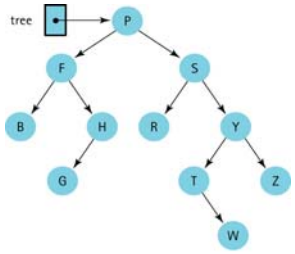
1. inorder (LNR)
2. postorder (LRN)
3. preorder (NLR).

Another technique is to move left to right from level to level.

- This algorithm is iterative, and its implementation involves using a queue.

Three Tree Traversals

1. inorder (LNR)
2. preorder (NLR
3. postorder (LRN)



Inorder: B F G H P R S T W Y Z
Preorder: P F B H G S R Y T W Z
Postorder: B G H F R W T Z Y S P

Recursive Btree Example

// algorithm for inorder traversal of the
// nodes in a given binary tree

if tree is NULL
  return
Else
  traverse the Left sub tree
  output the node value
  traverse the Right subtree
  return

Recursive Btree Example

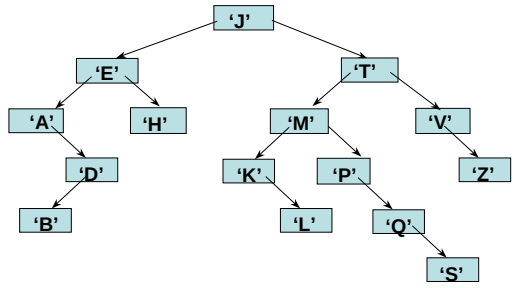
```
{ // in main
   traverseNodes(root);
}

void traverseNodes(btnode *tree)
// Recursive function that traverses the nodes
{
  if (tree = = NULL)
    return;
  else
      traverseNodes(tree->left);
      cout << tree -> nodeValue << endl;
      traverseNodes(tree->right);
      return;
}
```

A Binary Search Tree (BST) is . . .

A special kind of binary tree in which:

1. Each node contains a distinct data value,
2. The key values in the tree can be compared using "greater than" and "less than", and
3. The key value of each node in the tree is less than every key value in its right subtree, and greater than every key value in its left subtree.

Is 'F' in the binary search tree?



Binary Trees

- Most effective as a storage structure if it has high density
  - data are located on relatively short paths from the root.
  - A complete, full binary tree has the highest possible density
- Best used as a search structure in which inserts occur at the leaves and there are no deletes in the non leaf nodes

Matrices

- A Matrix is a two-dimensional array that corresponds to a row-column table of entries of a specified data type.
- Matrices are referenced using a pair of indices that specify the row and column location in the table.

	0	1	2	3
0	8	1	7	-2
1	0	-3	4	6
2	10	-14	1	0

Example:
The element mat[0][3] is -2
The element mat[1][2] is 4.

Energy Table

Years	Coal (0)	Gas (1)	Oil (2)	Hydro (3)	Nuclear (4)	Other (5)	
0	18.9	19.4	34.2	2.9	5.7	0.3	1989
1	19.1	19.3	33.6	3.0	6.2	0.2	1990
2	18.8	19.6	32.9	3.1	6.6	0.2	1991
3	18.9	20.3	33.5	2.8	6.7	0.2	1992
4	19.6	20.8	33.8	3.1	6.5	0.2	1993

Energy Sources

Energy Table

	Coal (0)	Gas (1)	Oil (2)	Hydro (3)	Nuclear (4)	Other (5)
0	18.9	19.4	34.2	2.9	5.7	0.3
1	19.1	19.3	33.6	3.0	6.2	0.2
2	18.8	19.6	32.9	3.1	6.6	0.2
3	18.9	20.3	33.5	2.8	6.7	0.2
4	19.6	20.8	33.8	3.1	6.5	0.2

Energy Sources

As a 2D Array

- Need to specify both the number of rows and columns during allocation
- Example:

```
const int ROWS = 5, COLS = 6;
double energyTable[ROWS][COLS];
```

- Typically processed like:

```
for (i = 0; i < ROWS;  i++)
{ //stuff before the inner loop
  for (j = 0; j < COLS; j++)
  {
   //inner loop operates on the [i,j]th element
  }
  // stuff after the  inner loop
}
```


Limitations

- C++ treats a matrix as an array of 1D arrays, of fixed size
  - No size attribute
- It is stored in memory by rows, one after the other
- Array elements cannot be correctly accessed unless the **compiler** knows the number of elements in each row (i.e. the number of columns)

Limitations

- When passing a 2D array as an argument you must specify a constant value for the number of columns
- These limitations, flexibility and potential growth/shrinkage requirements drive us to use vectors to implement a template-based matrix container
  - `vector <vector <T> > mat;`
  - `// the internal structure used`

A New Matrix Template* * Not in the STL

```
template <typename T>  class matrix
{
   private:  int nRows, nCols;
             vector <vector <T> > mat;
   public:
      // need constructors, accessor functions, resize
      // function, a new indexing operator, and
      // other common matrix operations as needed




}
```


Dynamic Matrix Template* * Not in the STL

```
template <typename T>  class matrix
{  private:
       int nRows, nCols;
       vector <vector <T> > mat;
   public:
       matrix (int numRows = 1, int numCols = 1,
             const T &initVal = T ( ) ):
             nRows (numRows), nCols (numCols),
        mat (numRows, vector <T> (numCols, initVal) )
       vector <T> & operator [ ] (int i); //index operator
       {
         if  (i < 0  | | i > nRows)
            throw indexRangeError (" matrix: invalid row #, i , nRows);
         return mat [i] ;
       }
       int getRows (  ) {return nRows;}
       int getCols (  ) {return nCols; }
       void resize (int numRows, int numCols);  // TBD
}
```


Using Your Matrix Template

- `matrix <double> energyTable  (5, 6, 0.0);`
- `matrix <Square> checkerBoard (8, 8);`
- `matrix <Pixel>  screen (1024, 768);`

EE 322C
Data Structures

Lecture 23

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

Lecture 26 Announcements

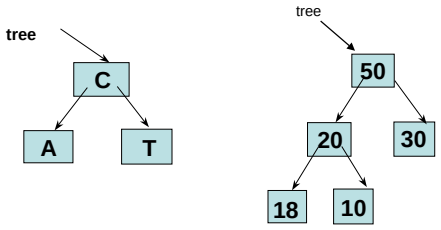
- Today's topics
  - Heaps
  - Graphs
- Extra credit homework
- Final exam coverage

What is a Heap?

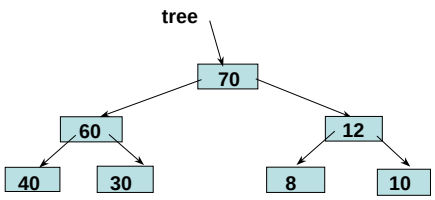
A heap is a binary tree that satisfies these special SHAPE and ORDER properties:

- Its shape must be a complete binary tree.
- Maximum Heap: For each node in the heap, the value stored in that node is greater than or equal to the value in each of its children.
- Minimum Heap: For each node in the heap, the value stored in that node is less than or equal to the value in each of its children.

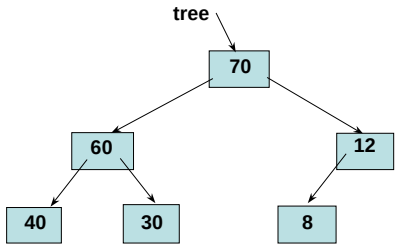
Are these Both Heaps?



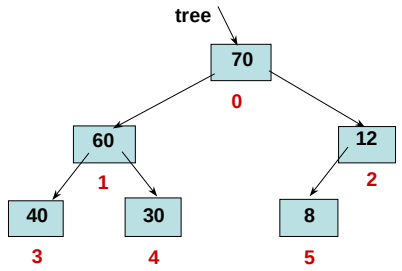
Is this a Heap?



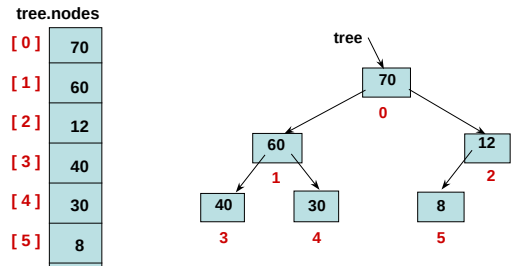
Where is the Largest Element in a Max Heap Always Found?



We Can Number the Nodes Left to Right by Level This Way



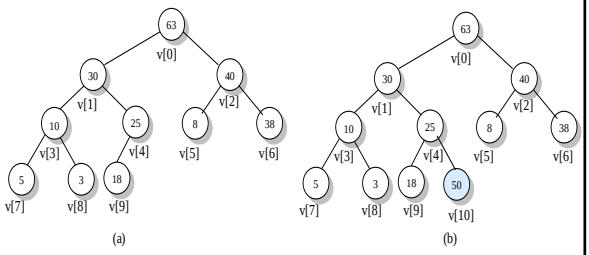
And use the Numbers as Indexes into A Vector to Store the Tree Nodes



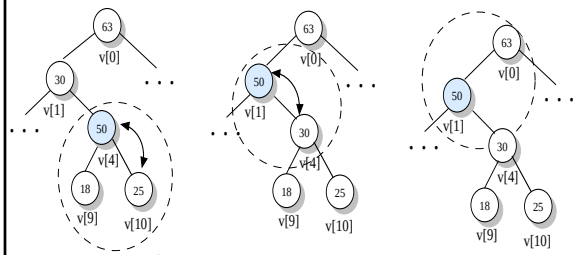
Common Heap Operations

- **Insertion**: place the new value at the back of the heap and filter it up the tree (aka push).
- **Deletion**: exchanging root value with the back of the heap and then filtering the new root down the tree,which now has one less element (aka pop)
  - Insert and delete running time: $O(\log_2 n)$
- **Heapifying**: apply the filter-down operation to the interior nodes, from the last interior node in the tree up to the root - running time: $O(n)$
- **Heapsort** : The $O(n \log_2 n)$ heapsort algorithm heapifies a vector and deletes repeatedly from the heap, putting each deleted value in its final position.

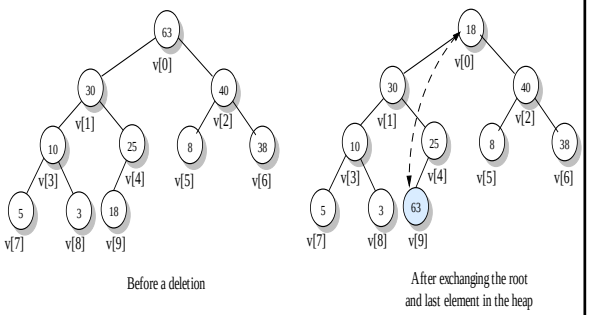
Heap Before and After Insertion of 50



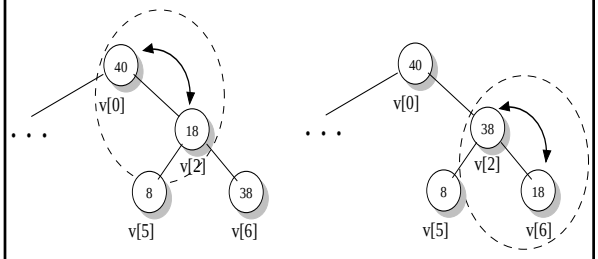
Reordering the tree in pushHeap()



Exchanging elements in popHeap( )



Adjusting the heap for popHeap()

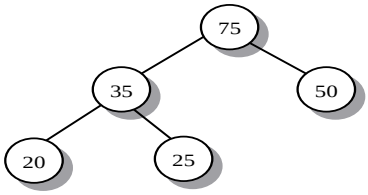


Step 1: Exchange 18 and 40

Step 2: Exchange 18 and 38

Implementing heap sort

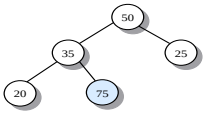
```
int arr[] = {50, 20, 75, 35, 25};
vector<int> h(arr, arr + 5);
```



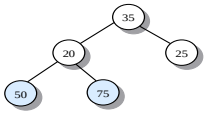
Heapified Tree

Implementing heap sort (Cont….)

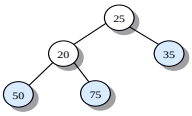
Calling popHeap() with last = 5 deletes 75 and stores it in h[4]



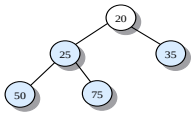
Calling popHeap() with last = 4 deletes 50 and stores it in h[3]



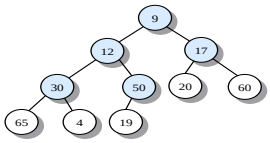
Calling popHeap() with last = 3 deletes 35 and stores it in h[2]



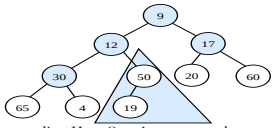
Calling popHeap() with last = 2 deletes 25 and stores it in h[1]



Heapifying a Vector

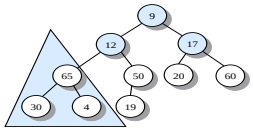


Initial Vector

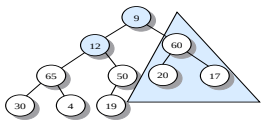


adjustHeap() at 4 causes no changes
(A)

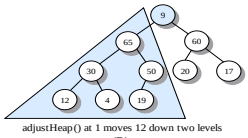
Heapifying a Vector (Cont…)



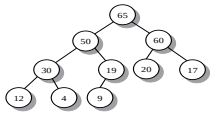
adjustHeap() at 3 moves 30 down
(B)



adjustHeap() at 2 moves 17 down
(C)



adjustHeap() at 1 moves 12 down two levels
(D)



adjustHeap() at 0 moves 9 down three levels
(E)

STL Template Functions

These assume that the heap is represented by a `sequence (vector)`

- `make_heap(iterator start,iterator end,Comp lessThanFcn)`
  - constructs a heap from a sequence
- `push_heap(start, end)`
  - pushes an element onto the end of the heap and rebuilds it
- `pop_heap(start, end)`
  - exchanges the first and last-1 elements and then rebuilds the heap
- `sort_heap(start, end)`
  - sorts a heap into descending order

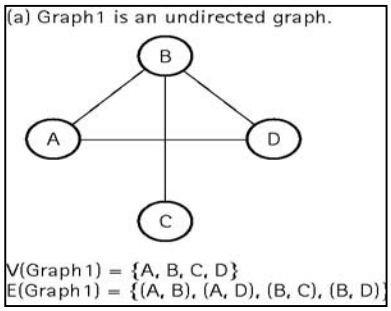
Graph Definitions

- **Graph:** A data structure that consists of a set of nodes and a set of edges that relate the nodes to each other
- **Vertex:** A node in a graph
- **Edge (arc):** A pair of vertices representing a connection between two nodes in a graph
- **Undirected graph:** A graph in which the edges have no direction
- **Directed graph (digraph):** A graph in which each edge is directed from one vertex to another (or the same) vertex

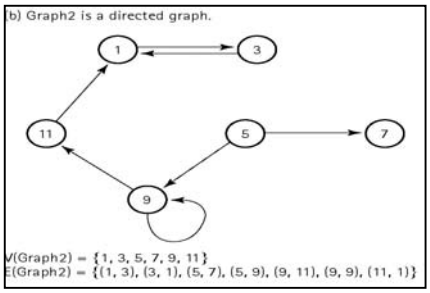
Formally

- A graph G is defined as follows:
  - G = (V,E)
- where
  - V(G) is a finite, nonempty set of vertices
  - E(G) is a set of edges
    - (written as pairs of vertices)

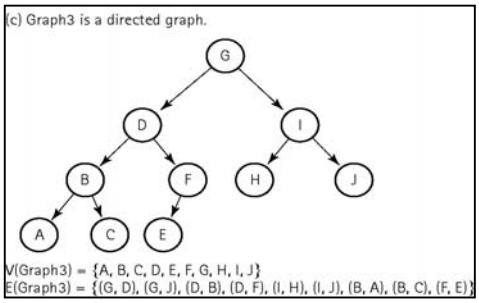
An Undirected Graph



A Directed Graph



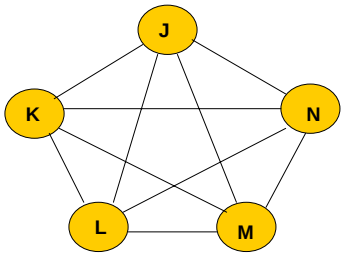
A Directed Graph



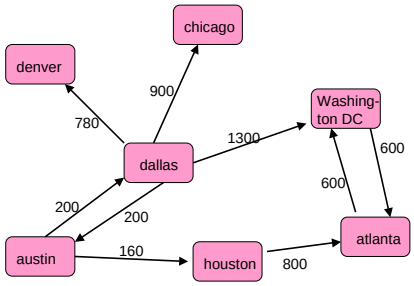
More Definitions

- **Adjacent vertices:** Two vertices in a graph that are connected by an edge
- **Path:** A sequence of vertices that connects two nodes in a graph
- **Complete graph:** A graph in which every vertex is directly connected to every other vertex
- **Weighted graph:** A graph in which each edge carries a value

A Complete Graph



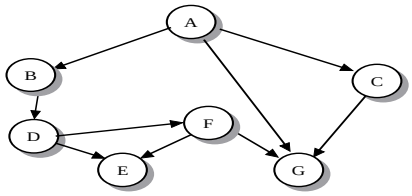
A Weighted Directed Graph



Definitions

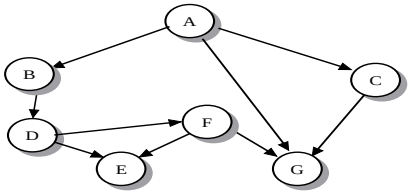
- **Depth-first search algorithm**: Visit all the nodes in a branch to its deepest point before moving up
- **Breadth-first search algorithm**: Visit all the nodes on one level before going to the next level
- **Single-source shortest-path algorithm**: An algorithm that displays the shortest path from a designated starting node to every other node in the graph (e.g, what is the shortest path from Austin to Atlanta in the previous graph)

Breadth-First Search



For example: A B G C D E F

Depth-First Search



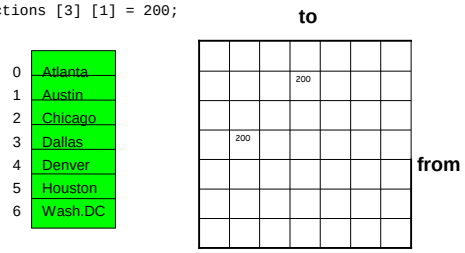
For example: A B D F G E C

Matrix-Based Implementation

- **Adjacency Matrix:** for a graph with N nodes, an N by N table that shows the existence (and weights) of all edges in the graph
- You can use the matrix template class to do this

Adjacency Matrix for Flight Connections

```
int N =  7;
vector <string>  cityNames (N);       // contains the cities
matrix  <int> cityConnections (N, N); // adjacency matrix
cityConnections [1] [3] = 200;
cityConnections [3] [1] = 200;
```

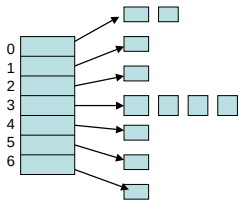


Linked Implementation

- **Adjacency List:** A linked list that identifies all the vertices to which a particular vertex is connected; each vertex has its own adjacency list

Adjacency List Representation of Graphs

```
int N =  numCities;
vector <cityNodes>  cityNet (N);

class  cityNode
{ string cityName;
  adjacencyNode  *nodelist;
}

class  adjacencyNode
{ int nodeNumber;
  int weight;
  adjacencyNode  *next;
}
```



Extra Credit Homework

- Basic rule: you have to do this on your own with no help from me, Matt or friends
- Use the adjacency list representation of graphs
  - See the example from the previous slide
- A *transitive closure* in a graph from a specific node is defined is defined to be all the nodes that can be reach from that specified node
  - It can reach all its adjacent nodes
  - It can reach all the adjacent nodes of its adjacent nodes, etc.

Extra Credit Homework

1) Define an undirected graph of 20 nodes using the adjacency list representation
2) Write a function that given a node in the graph returns a list of nodes.

Hints:

- Control structures
  - When is iteration appropriate?
  - When is recursion appropriate?
- Hard part: when to stop
  - Obviously, when you have done the whole graph
  - Need more: don't want to keep going indefinitely

Final Exam

- Topics
  - Software Engineering & object oriented principles
  - Classes, their structures etc
  - All the data structures
- Parts of the exam
  - True/False
  - Multiple Choice
  - Fill in the blank
  - What does this program do
  - Programming: data structures, functions

EE 322C
Data Structures

Lecture 24

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

Lecture 28 Announcements

- Exam (last exam, last day of class)
  - Wednesday 1 December 5:00 – 6:15 here
- Today's topics – more on associative containers (elements are stored by key and not by position)
  - More on Maps
  - Hashing

Map

- A map is a collection of key-value pairs that associate a key with a value.
  - In a map, there is only one value associated with a key.
- A map is often called an associative array because applying the index operator with the key as its argument accesses the associated value
- The map STL class has all the same operations found in set, however the elements are pairs not a single data item
- Balanced binary search tree is used for the STL implementation

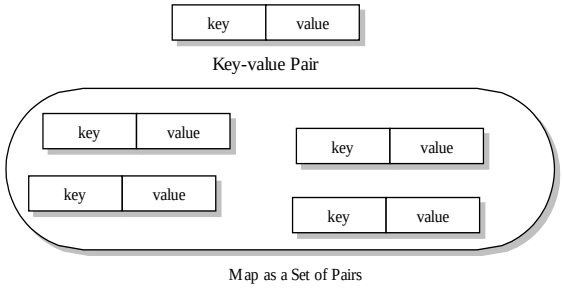
Key-Value Data

A map stores data as a key-value pair. In a pair, the *first* component is the key; the *second* is the value. Each component may have a different data type.

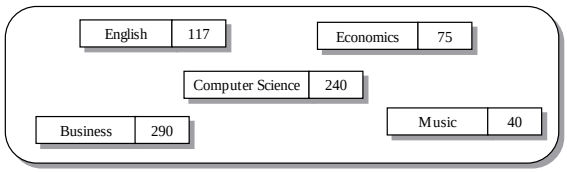
A lookup operation takes the key and returns an iterator that points to a matching (key, value) pair.

Only unique keys are allowed in maps

Maps



Map Example



degreeMajor: Map of string-int pairs

```
map <string, int> degreeMajor;
// assume values have been filled in as above
cout << degreeMajor [ "English" ];
degreeMajor [ "ECE" ] = 322;
```

Map Examples

```
/* this example creates a map of elements which are
   (capital letter, ascii value) pairs.  The program then
   accepts user input of a letter and finds and reports
   the associated ascii value of that letter if it is in the map.
*/
#include <iostream>
#include <map>
using namespace std;
int main()
{
   map<char, int> m;
   int i;
   // put (capital letter, ascii value) pairs into the map
   for (i =0; i < 26; i++)
   {   m.insert (pair <char, int> ('A' + i, 65 + i));
       //pair is a template struct with first&second components
   }
```


Map Examples

```
   char ch;
   cout << "enter the key character: ";
   cin >> ch;
   map <char, int>:: iterator p;
   // find the value given the key
   p = m.find (ch);
   if (p ! = m.end( ) )
       cout << "the ascii value of " << ch << " is "
       << p -> second;
   else
       cout << "the key of " << ch << " is not in the
   map";
   return 0;
}
```


Map Examples

```
/* this example creates a phone directory map in which the
   elements are (name, phone number) string pairs.  Several
   elements are inserted into the map, and then the user puts
   in a name and the program finds and reports the associated
   phone number if the name is in the map.
*/
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main()
{  map<string, string> directory;
   directory.insert(pair<string, string>("Tom", "555-4533"));
   directory.insert(pair<string, string>("Chris", "555-9678"));
   directory.insert(pair<string, string>("John", "555-8195"));
   directory.insert(pair<string, string>("Rachel", "555-0809"));
```


Map Examples

```
   string s;
   cout << "Enter name: ";
   cin >> s;
   map<string, string>::iterator p;
   p = directory.find(s);
   if(p != directory.end())
       cout << "Phone number: " << p->second;
   else
       cout << "Name not in directory.\n";
   return 0;
}
```

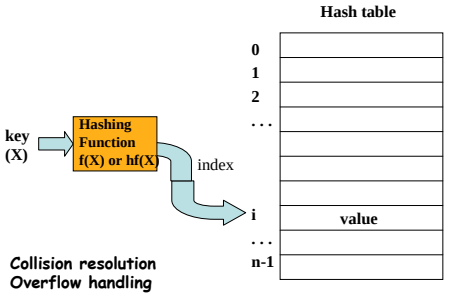

Hash Tables

- The hash table is organized as sequential storage divided into *b* buckets, each bucket with *s* slots. Each slot holds one element.
- The address of an identifier X in the table is gotten by computing some arithmetic function ~ f(X)
  - f(X) maps the set of possible identifiers onto the bucket numbers 0 to b-1; we will use the bucket # as the index
  - f(X) should be easy to compute and should spread out the elements to be stored - given a random value for X it should have an equal chance of hashing into any of the *b* buckets (uniformity)

Hash Tables

- Collision occurs when 2 different identifiers are hashed into the same bucket #
- Overflow occurs when a new identifier to be stored hashes into a full bucket
- If the bucket size is 1 then collision and overflow occurs simultaneously
- average time for a search of a hash table is O(1) - the worst case is O(n) where n is the total slots available
- Load factor a = m / (*s* * *b*), m is the # elements stored

Hashing Concept



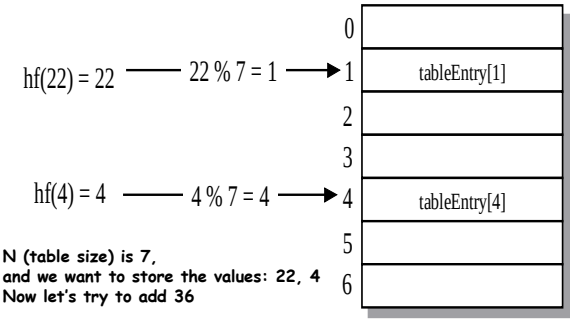
1

- Modulo division technique
- Let's assume that we wish to store integer values X in the hash table
  f (X) = X % N (N is the number of buckets)
- After obtaining an index by dividing the value from the hash function by the table size and taking the remainder, access the table.
- Normally, the number of elements in the table is much smaller than the number of possible data values, so collisions occur.
- To handle collisions, we must place a value that collides with an existing element into the table in such a way that we can efficiently find it later.

Hash Table Example

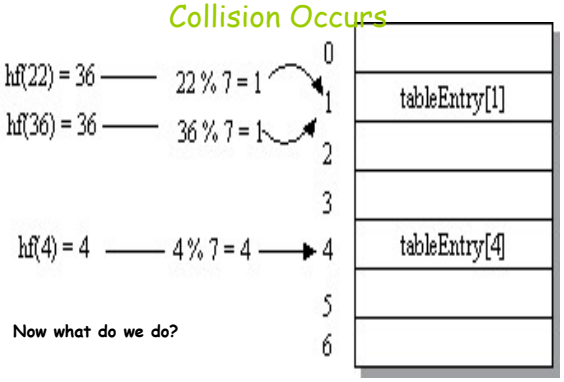
- Modulo division technique
- Let's assume that we wish to store integer values X in the hash table
  f (X) = X % N (N is the number of buckets)
- After obtaining an index by dividing the value from the hash function by the table size and taking the remainder, access the table.
- Normally, the number of elements in the table is much smaller than the number of possible data values, so collisions occur.
- To handle collisions, we must place a value that collides with an existing element into the table in such a way that we can efficiently find it later.

Example Hash Function



N (table size) is 7,
and we want to store the values: 22, 4
Now let's try to add 36

Collision Occurs



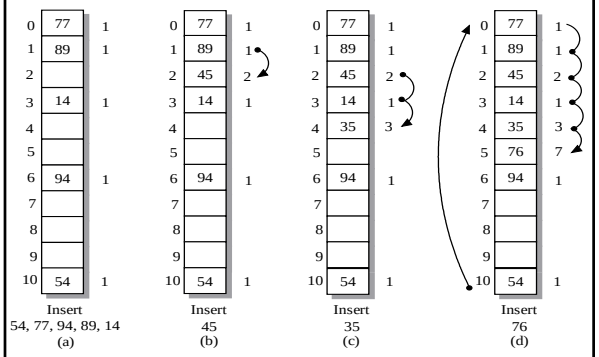
Now what do we do?

Collision Resolution

Linear open probe addressing

- The table is a vector or array of static size
- After using the hash function to compute a table index, look up the entry in the table.
- If the values match, perform operation as necessary.
- If the table entry is empty, insert the value in the table.
- Otherwise, probe forward circularly, looking for a match or an empty table slot.
- If the probe returns to the original starting point, the table is full.
- You can find table items that hashed to different table locations.
- Deleting an item is difficult in this scheme.

Hash Table: Linear Open Probe Addressing

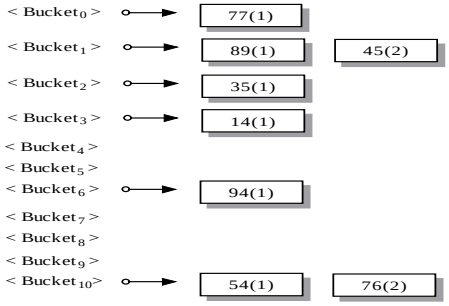


Collision Resolution

Chaining with separate lists.

- The hash table is a vector of list objects
- Each list (bucket) is a sequence of colliding items.
- After applying the hash function to compute the table index, search the list for the data value.
- If it is found, update its value; otherwise, insert the value at the back of the list.
- You search only items that collided at the same table location
- There is no limitation on the number of values in the table, and deleting an item from the table involves only erasing it from its corresponding list

Chaining with Separate Lists



Other Common Hash Functions

- Did Square
- Folding
- Digit analysis
- String combinations (use all characters)
- Custom hashing functions you design

Efficiency of Hash Methods

Hash table size = m, Number of elements in hash table = n, Load factor l = n / m

	Average Probes for Successful Search	Average Probes for Unsuccessful Search
Open Probe	$1+\frac{\lambda}{2}-\frac{1}{2m}$	$\frac{1}{2}+\frac{1}{2(1-\lambda)^2}$
Chaining	$\frac{1}{2}+\frac{1}{2(1-\lambda)^2}$	

Extra Credit Homework

- Basic rule: you have to do this on your own with no help from me, Matt or friends
- Use the adjacency list representation of graphs
  - See the example from the previous slide
- *A transitive closure* in a graph from a specific node is defined is defined to be all the nodes that can be reach from that specified node
  - It can reach all its adjacent nodes
  - It can reach all the adjacent nodes of its adjacent nodes, etc.

Extra Credit Homework

1) Define an undirected graph of 20 nodes using the adjacency list representation
2) Write a function that given a node in the graph returns a list of nodes.

Hints:

- Control structures
  - When is iteration appropriate?
  - When is recursion appropriate?
- Hard part: when to stop
  - Obviously, when you have done the whole graph
  - Need more: don't want to keep going indefinitely

Final Exam

- Topics
  - Software Engineering & object oriented principles
  - Classes, their structures etc
  - All the data structures
- Parts of the exam
  - True/False
  - Multiple Choice
  - Fill in the blank
  - What does this program do
  - Programming: data structures, functions

EE 322C
Data Structures

Last Exam

Fall 2004

perry@ece.utexas.edu
Office: ENS 623A
Office Hours: MW, 4:00- 5:00 pm

Last Exam - Totals

- 90-100 xxxxx xx
- 85+ xxxxx xxx
- 80+ xxxxx xxxxx xxxxx xxxxx
- 75+ xxxxx xxxxx x
- 70+ xxxxx x
- 65+ xxxxx
- 60+ xxxx
- Median: 80.5
- Mean: 79.5

Last Exam - Grades

- A 85 – 100 15
- B 80.5+ 20
- C 64+ 23
- D 63- 2

Last Exam – T/F

- 16 xxxxx xxxxx xxxxx xxxxx xxxxx xxxxx
- 14 xxxxx xxxxx xxxxx xxxxx xxxx
- 12 xxx
- 10 xx

- Median: 16
- Mean: 14.77

Last Exam - MC

- 21 xx
- 20.5 xxxxx
- 20 xxx
- 19.5 xxxxx x
- 19 xxxxx xxxxx xxx
- 18.5 xxxxx xxxxx
- 18 xxxx
- 17.5 xxxx
- 17 xxxxx
- 16.5 xx
- 16 xx
- 15.5 xxx
- 15 x

- Median: 18.5
- Mean: 18.45

Last Exam – Fill In

- 21 xxxxx xxxxx xxxxx xxxxx xxxxx xxxxx xx
- 20 xxxxx x
- 19 x
- 18 xxxxx xxx
- 17 xx
- 16 xxx
- 15 xx
- 14 xx
- 13.5 xx
- 13 xx

- Median: 21
- Mean 19.31

Last Exam – ID DS

- 16 xxxxx xxxxx xxxxx xxxxx xxx
- 14 xxxxx xxxx
- 12 xxxxx xxxxx xxxxx xx
- 11 x
- 10 xx
- 8 xxxxx x
- 6 x
- 4 x
- Median: 14
- Mean: 13.2

Last Exam – ID Function

- 17 x
- 16 xxxxx
- 15 xx
- 14 x
- 13 xxxx
- 12 xxxxx xxxxx x
- 11 xxxxx
- 10 xxxxx xxxxx xx
- 9 xxxxx xxxxx
- 7 xxxx
- 6 x
- 5 xxx
- 1 x
- Median: 10.5
- Mean: 10.67

Last Exam – Virtual Machine

- 9 xxxxx
- 8 xxx
- 6 xxx
- 5 xxxxx xx
- 4.5 x
- 4 xxxxx xx
- 3.5 x
- 3 xxxxx
- 2 xxxxx xxxxx x
- 1.5 xx
- 1 xxxxx xxx
- 0.5 xx
- 0 xxx
- Median: 3
- Mean: 3.62

Last Exam - Comments

- You all did fairly well on the first three parts – the parts that were basically memorization
- Slightly less well on identifying data structures
- Less well on understanding what the functions did
- And not very well on the virtual machine
  - Too much dancing around
  - Too much extraneous and erroneous stuff
  - Too little focus on what gets extended and what gets hidden