CSCI-1200 Data Structures — Spring 2017
Lecture 1 — Introduction to C++, STL, & Strings

Co-Instructors email: `ds_instructors@cs.rpi.edu`

Professor Herbert Holzbauer
Materials Research Center(MRC) 304 x8114
`holzbh@cs.rpi.edu`

Professor William Thompson
Amos Eaton(AE) 205 x6861
`thompw4@rpi.edu`

Today

- Discussion of Website & Syllabus:
  http://www.cs.rpi.edu/academics/courses/spring17/ds/
- Getting Started in C++ & STL, C++ Syntax, STL Strings

1.1 Transitioning from Python to C++ (from CSCI-1100 Computer Science 1)

- Python is a great language to learn the power and flexibility of programming and computational problem solving. This semester we will work in C++ and study lower level programming concepts, focusing on details including efficiency and memory usage.
- Outside of this class, when working on large programming projects, you will find it is not uncommon to use a mix of programming languages and libraries. The individual advantages of Python and C++ (and Java, and Perl, and C, and UNIX bash scripts, and ...) can be combined into an elegant (or terrifyingly complex) masterpiece.
- Here are a few excellent references recommended to help you transition from Python to C++:
  http://cs.slu.edu/~goldwasser/publications/python2cpp.pdf
  http://www4.wittenberg.edu/academics/mathcomp/shelburne/comp255/notes/Python2Cpp.pdf

1.2 Compiled Languages vs. Interpreted Languages

- C/C++ is a *compiled language*, which means your code is processed (compiled & linked) to produce a low-level machine language executable that can be run on your specific hardware. You must re-compile & re-link after you edit any of the files – although a smart development environment or `Makefile` will figure out what portions need to be recompiled and save some time (especially on large programming projects with many lines of code and many files). Also, if you move your code to a different computer you will usually need to recompile. Generally the extra work of compilation produces an efficient and optimized executable that will run fast.
- In contrast, many newer languages including Python, Java, & Perl are *interpreted languages*, that favor incremental development where you can make changes to your code and immediately run all or some of your code without waiting for compilation. However, an interpreted program will often run slower than a compiled program.
- These days, the process of compilation is almost instantaneous for simple programs, and in this course we encourage you to follow the same incremental editing & frequent testing development strategy that is employed with interpreted languages.
- Finally, many interpreted languages have a Just-In-Time-Compiler (JIT) that can run an interpreted programming language and perform optimization on-the-fly resulting in program performance that rivals optimized compiled code. Thus, the differences between compiled and interpreted languages are somewhat blurry.
- You will practice the cycle of coding & compilation & testing during Lab 1. You are encouraged to try out different development environments (code editor & compiler) and quickly settle on one that allows you to be most productive. Ask the your lab TAs & mentors about their favorite programming environments! The course website includes many helpful links as well.
- As you see in today's handout, C++ has more required punctuation than Python, and the syntax is more restrictive. The compiler will proofread your code in detail and complain about any mistakes you make. Even long-time C++ programmers make mistakes in syntax, and with practice you will become familiar with the compiler's error messages and how to correct your code.

1.3 A Sample C++ Program: Find the Roots of a Quadratic Polynomial

```
#include <iostream>   // library for reading & writing from the console/keyboard
#include <cmath>      // library with the square root function & absolute value
#include <cstdlib>    // library with the exit function


// Returns true if the candidate root is indeed a root of the polynomial a*x*x + b*x + c = 0
bool check_root(int a, int b, int c, float root) {
  // plug the value into the formula
  float check = a * root * root + b * root + c;
  // see if the absolute value is zero (within a small tolerance)
  if (fabs(check) > 0.0001) {
    std::cerr << "ERROR:  " << root << " is not a root of this formula." << std::endl;
    return false;
  } else {
    return true;
  }
}

/* Use the quadratic formula to find the two real roots of polynomial.   Returns
true if the roots are real, returns false if the roots are imaginary.  If the roots
are real, they are returned through the reference parameters root_pos and root_neg. */
bool find_roots(int a, int b, int c, float &root_pos, float &root_neg) {
  // compute the quantity under the radical of the quadratic formula
  int radical = b*b - 4*a*c;
  // if the radical is negative, the roots are imaginary
  if (radical < 0) {
    std::cerr << "ERROR:  Imaginary roots" << std::endl;
    return false;
  }
  float sqrt_radical = sqrt(radical);
  // compute the two roots
  root_pos = (-b + sqrt_radical) / float(2*a);
  root_neg = (-b - sqrt_radical) / float(2*a);
  return true;
}

int main() {
  // We will loop until we are given a polynomial with real roots
  while (true) {
    std::cout << "Enter 3 integer coefficients to a quadratic function: a*x*x + b*x + c = 0" << std::endl;
    int my_a, my_b, my_c;
    std::cin >> my_a >> my_b >> my_c;
    // create a place to store the roots
    float root_1, root_2;
    bool success = find_roots(my_a,my_b,my_c, root_1,root_2);
    // If the polynomial has imaginary roots, skip the rest of this loop and start over
    if (!success) continue;
    std::cout << "The roots are: " << root_1 << " and " << root_2 << std::endl;
    // Check our work...
    if (check_root(my_a,my_b,my_c, root_1) && check_root(my_a,my_b,my_c, root_2)) {
      // Verified roots, break out of the while loop
      break;
    } else {
      std::cerr << "ERROR:  Unable to verify one or both roots." << std::endl;
      // if the program has an error, we choose to exit with a
      // non-zero error code
      exit(1);
    }
  }
  // by convention, main should return zero when the program finishes normally
  return 0;
}
```

1.4 Some Basic C++ Syntax

- Comments are indicated using `//` for single line comments and `/*` and `*/` for multi-line comments.
- `#include` asks the compiler for parts of the standard library and other code that we wish to use (e.g. the input/output stream function `std::cout`).
- `int main()` is a necessary component of all C++ programs; it returns a value (integer in this case) **and** it may have parameters.
- `{ }`: the curly braces indicate to C++ to treat *everything* between them as a unit.

1.5 The C++ Standard Library, a.k.a. "STL"

- The standard library contains types and functions that are important extensions to the core C++ language. We will use the standard library to such a great extent that it will feel like part of the C++ core language. `std` is a *namespace* that contains the standard library.
- I/O streams are the first component of the standard library that we see. `std::cout` ("console output") and `std::endl` ("end line") are defined in the standard library header file, `iostream`

1.6 Variables and Types

- A *variable* is an object with a name. A name is a C++ identifier such as "`a`", "`root_1`", or "`success`".
- An *object* is computer memory that has a type. A type (e.g., `int`, `float`, and `bool`) is a memory structure and a set of operations.
- For example, `float` is a type and each `float` variable is assigned to 4 bytes of memory, and this memory is formatted according IEEE floating point standards for what represents the exponent and mantissa. There are many operations defined on floats, including addition, subtraction, printing to the screen, etc.
- In C++ and Java the programmer must specify the data type when a new variable is declared. The C++ compiler enforces type checking (a.k.a. *static typing*). In contrast, the programmer does not specify the type of variables in Python and Perl. These languages are *dynamically-typed* — the interpreter will deduce the data type at runtime.

1.7 Expressions, Assignments and Statements

Consider the *statement*: `root_pos = (-b + sqrt_radical) / float(2*a);`

- The calculation on the right hand side of the `=` is an expression. You should review the definition of C++ arithmetic expressions and operator precedence from any reference textbook. The rules are pretty much the same in C++ and Java and Python.
- The value of this expression is assigned to the memory location of the float variable `root_pos`. Note also that if all expression values are type `int` we need a *cast* from `int` to `float` to prevent the truncation of integer division.
- The *float(2*a)* expression casts the integer value *2*a* to the proper float representation.

1.8 Conditionals and IF statements

- The general form of an if-else statement is

```
if (conditional-expression)
   statement;
else
   statement;
```

- Each `statement` may be a single statement, such as the `cout` statement above, a structured statement, or a compound statement delimited by {...}.

1.9 Functions and Arguments

- Functions are used to:
  - Break code up into modules for ease of programming and testing, and for ease of reading by other people (never, ever, under-estimate the importance of this!).
  - Create code that is reusable at several places in one program and by several programs.
- Each function has a sequence of parameters and a return type. The function *prototype* below has a return type of `bool` and five parameters.

```
bool find_roots(int a, int b, int c, float &root_pos, float &root_neg);
```

- The order and types of the parameters in the calling function (the main function in this example) must match the order and types of the parameters in the function prototype.

1.10 Value Parameters and Reference Parameters

- What's with the `&` symbol on the 4th and 5th parameters in the `find_roots` function prototype?
- Note that when we call this function, we haven't yet stored anything in those two root variables.

```
float root_1, root_2;
bool success = find_roots(my_a,my_b,my_c, root_1,root_2);
```

- The first first three parameters to this function are *value parameters*.
  - These are essentially local variables (in the function) whose initial values are *copies* of the values of the corresponding argument in the function call.
  - Thus, the value of `my_a` from the main function is used to initialize `a` in function `find_roots`.
  - Changes to value parameters within the called function do NOT change the corresponding argument in the calling function.
- The final two parameters are *reference parameters*, as indicated by the `&`.
  - Reference parameters are just aliases for their corresponding arguments. No new objects are created.
  - As a result, changes to reference parameters are changes to the corresponding variables (arguments) in the calling function.
- In general, the "Rules of Thumb" for using value and reference parameters:
  - When a function (e.g., `check_root`) needs to provide just one simple result, make that result the return value of the function and pass other parameters by value.
  - When a function needs to provide more than one result (e.g., `find_roots`, these results should be returned using multiple reference parameters.
- We'll see more examples of the importance of value vs. reference parameters as the semester continues.

1.11 `for` & `while` Loops

- Here is the basic form of a `for` loop:

```
for (expr1; expr2; expr3)
   statement;
```

  - `expr1` is the initial expression executed at the start before the loop iterations begin;
  - `expr2` is the test applied before the beginning of each loop iteration, the loop ends when this expression evaluates to `false` or `0`;
  - `expr3` is evaluated at the very end of each iteration;
  - `statement` is the "loop body"
- Here is the basic form of a `while` loop:

```
while (expr)
   statement;
```

  `expr` is checked before entering the loop and after each iteration. If `expr` ever evaluates the false the loop is finished.

1.12 C-style Arrays

- An array is a fixed-length, consecutive sequence of objects all of the same type. The following declares an array with space for 15 double values. Note the spots in the array are currently *uninitialized*.

```
double a[15];
```

- The values are accessed through subscripting operations. The following code assigns the value 3.14159 to location `i=5` of the array. Here `i` is the *subscript* or *index*.

```
int i = 5;
a[i] = 3.14159;
```

- In C/C++, array indexing starts at 0.
- Arrays are fixed size, and each array knows *NOTHING* about its own size. The programmer must keep track of the size of each array. (Note: C++ STL has generalization of C-style arrays, called *vectors*, which do not have these restrictions. More on this in Lecture 2!)

1.13 Python Strings vs. C `chars` vs. C-style Strings vs. C++ STL Strings

- Strings in Python are immutable, and there is no difference between a string and a char in Python. Thus, `'a'` and `"a"` are both strings in Python, not individual characters. In C++ & Java, single quotes create a character type (exactly one character) and double quotes create a string of 0, 1, 2, or more characters.
- A "C-style" string is an array of `char`s that ends with the special char '\0'. C-style strings (`char*` or `char[]`) can be edited, and there are a number of helper functions to help with common operations. However...
- The "C++-style" STL `string` type has a wider array of operations and functions, which are more convenient and more powerful.

1.14 About STL String Objects

- A `string` is an object type defined in the standard library to contain a sequence of characters.
- The string type, like all types (including `int`, `double`, `char`, `float`), defines an interface, which includes construction (initialization), operations, functions (methods), and even other types(!).
- When an object is created, a special function is run called a "constructor", whose job it is to initialize the object. There are several ways of constructing string objects:
  - By default to create an empty string: `std::string my_string_var;`
  - With a specified number of instances of a single char: `std::string my_string_var2(10, ' ');`
  - From another string: `std::string my_string_var3(my_string_var2);`
- The notation `my_string_var.size()` is a call to a function `size` that is defined as a **member function** of the `string` class. There is an equivalent member function called `length`.
- Input to string objects through streams (e.g. reading from the keyboard or a file) includes the following steps:
  1. The computer inputs and discards white-space characters, one at a time, until a non-white-space character is found.
  2. A sequence of non-white-space characters is input and stored in the string. This overwrites anything that was already in the string.
  3. Reading stops either at the end of the input or upon reaching the next white-space character (without reading it in).
- The (overloaded) operator '+' is defined on strings. It concatenates two strings to create a third string, without changing either of the original two strings.
- The assignment operation '=' on strings overwrites the current contents of the string.
- The individual characters of a string can be accessed using the subscript operator `[]` (similar to arrays).
  - Subscript 0 corresponds to the first character.
  - For example, given `std::string a = "Susan";`
    Then `a[0] == 'S'` and `a[1] == 'u'` and `a[4] == 'n'`.

- Strings define a special type `string::size_type`, which is the type returned by the string function `size()` (and `length()`).
  - The `::` notation means that `size_type` is defined within the scope of the `string` type.
  - `string::size_type` is generally equivalent to `unsigned int`.
  - You may see have compiler warnings and potential compatibility problems if you compare an `int` variable to `a.size()`.

This seems like a lot to remember. Do I need to memorize this? Where can I find all the details on `string` *objects?*

1.15 Problem: Writing a Name Along a Diagonal

- Let's study a simple program to read in a name using `std::cin` and then output a fancier version to `std::cout`, written along a diagonal inside a box of asterisks. Here's how the program should behave:

```
What is your first name? Bob

*******
*     *
* B   *
*  o  *
*   b *
*     *
*******
```

- There are two main difficulties:
  - Making sure that we can put the characters in the right places on the right lines.
  - Getting the asterisks in the right positions and getting the right number of blanks on each line.

```
#include <iostream>
#include <string>

int main() {
  std::cout << "What is your first name? ";
  std::string first;
  std::cin >> first;
  const std::string star_line(first.size()+4, '*');
  std::string middle_line = "*" + std::string(first.size()+2,' ') + "*";
  std::cout << '\n' << star_line << '\n' << middle_line << std::endl;
  // Output the interior of the greeting, one line at a time.
  for (unsigned int i = 0; i < first.size(); ++i ) {
    // Create the output line by overwriting a single character from the
    // first name in location i+2.  After printing it restore the blank.
    middle_line[ i+2 ] = first[i];
    std::cout << middle_line << '\n';
    middle_line[ i+2 ] = ' ';
  }
  std::cout << middle_line << '\n' << star_line << std::endl;
  return 0;
}
```

CSCI-1200 Data Structures — Spring 2017
Collaboration Policy & Academic Integrity

iClicker Lecture exercises

Responses to iClicker lecture exercises will be used to earn incentives for the Data Structures course. Discussion of collaborative iClicker lecture exercises with those seated around you is encouraged. However, if we find anyone using an iClicker that is registered to another individual or using more than one iClicker, we will confiscate all iClickers involved and report the incident to the Dean of Students.

Academic Integrity for Exams

All exams for this course will be completed individually. Copying, communicating, or using disallowed materials during an exam is cheating, of course. Students caught cheating on an exam will receive an F in the course and will be reported to the Dean of Students for further disciplinary action.

Collaboration Policy for Programming Labs

Collaboration is encouraged during the weekly programming labs. Students are allowed to talk through and assist each other with these programming exercises. Students may ask for help from each other, the graduate lab TA, and undergraduate programming mentors. But each student must write up and debug their own lab solutions on their own laptop and be prepared to present and discuss this work with the TA to receive credit for each checkpoint.

As a general guideline, students may look over each other's shoulders at their labmate's laptop screen during lab — this is the best way to learn about IDEs, code development strategies, testing, and debugging. However, looking should not lead to line-by-line copying. Furthermore, each student should retain control of their own keyboard. While being assisted by a classmate or a TA, the student should remain fully engaged on problem solving and ask plenty of questions. Finally, other than the specific files provided by the instructor, electronic files or file excerpts should not be shared or copied (by email, text, Dropbox, or any other means).

Homework Collaboration Policy

Academic integrity is a complicated issue for individual programming assignments, but one we take very seriously. Students naturally want to work together, and it is clear they learn a great deal by doing so. Getting help is often the best way to interpret error messages and find bugs, even for experienced programmers. Furthermore, in-depth discussions about problem solving, algorithms, and code efficiency are invaluable and make us all better software engineers. In response to this, the following rules will be enforced for programming assignments:

- Students may read through the homework assignment together and discuss what is asked by the assignment, examples of program input & expected output, the overall approach to tackling the assignment, possible high level algorithms to solve the problem, and recent concepts from lecture that might be helpful in the implementation.

- Students are not allowed to work together in writing code or pseudocode. Detailed algorithms and implementation must be done individually. Students may not discuss homework code in detail (line-by-line or loop-by-loop) while it is being written or afterwards. In general, students should not look at each other's computer screen (or hand-written or printed assignment design notes) while working on homework. As a guideline, if an algorithm is too complex to describe orally (without dictating line-by-line), then sharing that algorithm is disallowed by the homework collaboration policy.

- Students are allowed to ask each other for help in interpreting error messages and in discussing strategies for testing and finding bugs. First, ask for help orally, by describing the symptoms of the problem. For each homework, many students will run into similar problems and after hearing a general description of a problem, another student might have suggestions for what to try to further diagnose or fix the issue. If that doesn't work, and if the compiler error message or flawed output is particularly lengthy, it is okay to ask another student to briefly look at the computer screen to see the details of the error message and the corresponding line of code. Please see a TA during office hours if a more in-depth examination of the code is necessary.

- Students may not share or copy code or pseudocode. Homework files or file excerpts should never be shared electronically (by email, text, LMS, Dropbox, etc.). Homework solution files from previous years

(either instructor or student solutions) should not be used in any way. Students must not leave their code (either electronic or printed) in publicly-accessible areas. Students may not share computers in any way when there is an assignment pending. Each student is responsible for securing their homework materials using all reasonable precautions. These precautions include: Students should password lock the screen when they step away from their computer. Homework files should only be stored on private accounts/computers with strong passwords. Homework notes and printouts should be stored in a locked drawer/room.

- Students may not show their code or pseudocode to other students as a means of helping them. Well-meaning homework help or tutoring can turn into a violation of the homework collaboration policy when stressed with time constraints from other courses and responsibilities. Sometimes good students who feel sorry for struggling students are tempted to provide them with "just a peek" at their code. Such "peeks" often turn into extensive copying, despite prior claims of good intentions.
- Students may not receive detailed help on their assignment code or pseudocode from individuals outside the course. This restriction includes tutors, students from prior terms, friends and family members, internet resources, etc.
- All collaborators (classmates, TAs, ALAC tutors, upperclassmen, students/instructor via LMS, etc.), and all of the resources (books, online reference material, etc.) consulted in completing this assignment must be listed in the README.txt file submitted with the assignment.

These rules are in place for each homework assignment and extends two days after the submission deadline.

Homework Plagiarism Detection and Academic Dishonesty Penalty

We use an automatic code comparison tool to help spot homework assignments that have been submitted in violation of these rules. The tool takes all assignments from all sections and all prior terms and compares them, highlighting regions of the code that are similar. The plagiarism tool looks at core code structure and is not fooled by variable and function name changes or addition of comments and whitespace.

The instructor checks flagged pairs of assignments very carefully, to determine which students may have violated the rules of collaboration and academic integrity on programming assignments. When it is believed that an incident of academic dishonesty has occurred, the involved students are contacted and a meeting is scheduled. All students caught cheating on a programming assignment (both the copier and the provider) will be punished. For undergraduate students, the standard punishment for the first offense is a 0 on the assignment and a full letter grade reduction on the final semester grade. Students whose violations are more flagrant will receive a higher penalty. Undergraduate students caught a second time will receive an immediate F in the course, regardless of circumstances. Each incident will be reported to the Dean of Students.

Graduate students found to be in violation of the academic integrity policy for homework assignments on the first offense will receive an F in the course and will be reported both to the Dean of Students and to the chair of their home department with the strong advisement that they be ineligible to serve as a teaching assistant for any course at RPI.

Academic Dishonesty in the Student Handbook

Refer to the *The Rensselaer Handbook of Student Rights and Responsibilities* for further discussion of academic dishonesty. Note that: "Students found in violation of the academic dishonesty policy are prohibited from dropping the course in order to avoid the academic penalty."

Number of Students Found in Violation of the Policy

Historically, 5-10% of students are found to be in violation of the academic dishonesty policy each semester. Many of these students immediately admit to falling behind with the coursework and violating one or more of the rules above and if it is a minor first-time offense may receive a reduced penalty.

Read this document in its entirety. If you have any questions, contact the instructor or the TAs immediately. Sign this form and give it to your TA during your first lab section.

Name: **Section #:**

Signature: **Date:**

CSCI-1200 Data Structures — Spring 2017
Lecture 2 — STL Strings & Vectors

Announcements

- HW 1 will be available on-line this afternoon through the website (on the "Calendar").
- Be sure to read through this information as you start implementation of HW1:
  "Misc Programming Information" (a link at the bottom of the left bar of the website).
- TA & instructor office hours are posted on website ("Weekly Schedule").
- If you have not resolved issues with the C++ environment on your laptop, please do so immediately.
- If you cannot access Piazza or the homework submission server, please email the instructor ASAP with your RCS ID and section number.
- Because many students were dealing with lengthy compiler/editor installation, registration confusion, etc., we will allow (for the first lab only!) students to get checked off for any remaining Lab 1 checkpoints at the beginning of next week's Lab 2 or in your grad TA's normal office hours.

Today

- STL Strings, `char` arrays (C-style Strings), & converting between these two types
- L-values vs. R-values
- STL Vectors as "smart arrays"

2.1 String Concatenation and Creation of Temporary String Object

- The following statement creates a new string by "adding" (concatenating) other strings together:

```
std::string my_line = "*" + std::string(first.size()+2,' ') + "*";
```

- The expression `std::string(first.size()+2, ' ')` within this statement creates a temporary STL string but does not associate it with a variable.

2.2 Character Arrays and String Literals

- In the line below `"Hello!"` is a *string literal* and it is also an array of characters (with no associated variable name).

```
cout << "Hello!" << endl;
```

- A char array can be initialized as: `char h[] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};`

  or as: `char h[] = "Hello!";`

  In either case, array `h` has 7 characters, the last one being the null character.
- The C language provides many functions for manipulating these "C-style strings". We don't study them much anymore because the "C++ style" STL string library is much more logical and easier to use. If you want to find out more about functions for C-style strings look at the cstdlib library `http://www.cplusplus.com/reference/cstdlib/`.
- One place we do use them is in file names and command-line arguments, which you will use in Homework 1.

2.3 Conversion Between Standard Strings and C-Style String Literals

- We regularly convert/cast between C-style & C++-style (STL) strings. For example:

```
std::string s1( "Hello!" );
std::string s2( h );
```

  where `h` is as defined above.
- You can obtain the C-style string from a standard string using the member function `c_str`, as in `s1.c_str()`.

2.4 L-Values and R-Values

- Consider the simple code below. String `a` becomes `"Tim"`. No big deal, right? Wrong!

```
std::string a = "Kim";
std::string b = "Tom";
a[0] = b[0];
```

- Let's look closely at the line: `a[0] = b[0];` and think about what happens.

  In particular, what is the difference between the use of `a[0]` on the left hand side of the assignment statement and `b[0]` on the right hand side?

- Syntactically, they look the same. But,
  - The expression `b[0]` gets the char value, `'T'`, from string location 0 in `b`. This is an *r-value*.
  - The expression `a[0]` gets a reference to the memory location associated with string location 0 in `a`. This is an *l-value*.
  - The assignment operator stores the value in the referenced memory location.

  The difference between an *r-value* and an *l-value* will be especially significant when we get to writing our own operators later in the semester

- What's wrong with this code?

```
std::string foo = "hello";
foo[2] = 'X';
cout << foo;
'X' = foo[3];
cout << foo;
```

  Your C++ compiler will complain with something like: "`non-lvalue in assignment`"

2.5 Standard Template Library (STL) Vectors: Motivation

- Example Problem: Read an unknown number of grades and compute some basic statistics such as the *mean* (average), *standard deviation*, *median* (middle value), and *mode* (most frequently occurring value).
- Our solution to this problem will be much more elegant, robust, & less error-prone if we use the STL `vector` class. Why would it be more difficult/wasteful/buggy to try to write this using C-style (dumb) arrays?

2.6 STL Vectors: a.k.a. "C++-Style", "Smart" Arrays

- Standard library "container class" to hold sequences.
- A vector acts like a dynamically-sized, one-dimensional array.
- Capabilities:
  - Holds objects of any type
  - Starts empty unless otherwise specified
  - Any number of objects may be added to the end — there is no limit on size.
  - It can be treated like an ordinary array using the subscripting operator.
  - A vector knows how many elements it stores! (unlike C arrays)
  - There is NO automatic checking of subscript bounds.
- Here's how we create an empty vector of integers:

```
std::vector<int> scores;
```

- Vectors are an example of a *templated container class*. The angle brackets `< >` are used to specify the type of object (the "template type") that will be stored in the vector.

- `push_back` is a vector function to append a value to the end of the vector, increasing its size by one. This is an $O(1)$ operation (on average).
  - There is NO corresponding `push_front` operation for vectors.
- `size` is a function defined by the vector type (the vector class) that returns the number of items stored in the vector.
- After vectors are initialized and filled in, they may be treated *just like arrays.*
  - In the line

    ```
    sum += scores[i];
    ```

    `scores[i]` is an "r-value", accessing the value stored at location `i` of the vector.
  - We could also write statements like

    ```
    scores[4] = 100;
    ```

    to change a score. Here `scores[4]` is an "l-value", providing the means of storing 100 at location 4 of the vector.
  - It is the job of the programmer to ensure that any subscript value i that is used is legal —- at least 0 and strictly less than `scores.size()`.

2.7 Initializing a Vector — The Use of Constructors

Here are several different ways to initialize a vector:

- This "constructs" an empty vector of integers. Values must be placed in the vector using `push_back`.

  ```
  std::vector<int> a;
  ```

- This constructs a vector of 100 doubles, each entry storing the value 3.14. New entries can be created using `push_back`, but these will create entries 100, 101, 102, etc.

  ```
  int n = 100;
  std::vector<double> b( 100, 3.14 );
  ```

- This constructs a vector of 10,000 ints, but provides no initial values for these integers. Again, new entries can be created for the vector using `push_back`. These will create entries 10000, 10001, etc.

  ```
  std::vector<int> c( n*n );
  ```

- This constructs a vector that is an exact copy of vector `b`.

  ```
  std::vector<double> d( b );
  ```

- This is a compiler error because no constructor exists to create an int vector from a double vector. These are different types.

  ```
  std::vector<int> e( b );
  ```

2.8 Exercises

1. After the above code constructing the three vectors, what will be output by the following statement?

   ```
   cout << a.size() << endl << b.size() << endl << c.size() << endl;
   ```

2. Write code to construct a vector containing 100 doubles, each having the value 55.5.
3. Write code to construct a vector containing 1000 doubles, containing the values 0, 1, $\sqrt{2}$, $\sqrt{3}$, $\sqrt{4}$, $\sqrt{5}$, etc. Write it two ways, one that uses `push_back` and one that does not use `push_back`.

2.9 Example: Using Vectors to Compute Standard Deviation

Definition: If $a_0, a_1, a_2, \ldots, a_{n-1}$ is a sequence of n values, and μ is the average of these values, then the standard deviation is

$$\left[\frac{\sum_{i=0}^{n-1}(a_i-\mu)^2}{n-1}\right]^{\frac{1}{2}}$$

```
// Compute the average and standard deviation of an input set of grades.
#include <fstream>
#include <iomanip>
#include <iostream>
#include <vector>          // to access the STL vector class
#include <cmath>           // to use standard math library and sqrt

int main(int argc, char* argv[]) {
  if (argc != 2) {
    std::cerr << "Usage: " << argv[0] << " grades-file\n";
    return 1;
  }
  std::ifstream grades_str(argv[1]);
  if (!grades_str.good()) {
    std::cerr << "Can not open the grades file " << argv[1] << "\n";
    return 1;
  }
  std::vector<int> scores;  // Vector to hold the input scores; initially empty.
  int x;                    // Input variable

  // Read the scores, appending each to the end of the vector
  while (grades_str >> x) {
    scores.push_back(x);
  }

  // Quit with an error message if too few scores.
  if (scores.size() == 0) {
    std::cout << "No scores entered.  Please try again!" << std::endl;
    return 1;   // program exits with error code = 1
  }

  // Compute and output the average value.
  int sum = 0;
  for (unsigned int i = 0; i < scores.size(); ++ i) {
    sum += scores[i];
  }
  double average = double(sum) / scores.size();
  std::cout << "The average of " << scores.size() << " grades is "
            << std::setprecision(3) << average << std::endl;

  // Exercise:  compute and output the standard deviation.
  double sum_sq_diff = 0.0;
  for (unsigned int i=0; i<scores.size(); ++i) {
    double diff = scores[i] - average;
    sum_sq_diff += diff*diff;
  }
  double std_dev = sqrt(sum_sq_diff / (scores.size()-1));
  std::cout << "The standard_deviation of " << scores.size()
            << " grades is " << std::setprecision(3) << std_dev << std::endl;

  return 0;   // everything ok
}
```

2.10 Standard Library Sort Function

- The standard library has a series of algorithms built to apply to container classes.
- The prototypes for these algorithms (actually the functions implementing these algorithms) are in header file `algorithm`.
- One of the most important of the algorithms is `sort`.
- It is accessed by providing the beginning and end of the container's interval to sort.

- As an example, the following code reads, sorts and outputs a vector of doubles:

```
double x;
std::vector<double> a;
while (std::cin >> x)
   a.push_back(x);
std::sort(a.begin(), a.end());
for (unsigned int i=0; i < a.size(); ++i)
   std::cout << a[i] << '\n';
```

- `a.begin()` is an *iterator* referencing the first location in the vector, while `a.end()` is an *iterator* referencing one past the last location in the vector.
  - We will learn much more about iterators in the next few weeks.
  - Every container has iterators: strings have `begin()` and `end()` iterators defined on them.
- The ordering of values by `std::sort` is least to greatest (technically, non-decreasing). We will see ways to change this.

2.11 Example: Computing the Median

The median value of a sequence is less than half of the values in the sequence, and greater than half of the values in the sequence. If $a_0, a_1, a_2, \ldots, a_{n-1}$ is a sequence of n values AND if the sequence is sorted such that $a_0 \leq a_1 \leq a_2 \leq \cdots \leq a_{n-1}$ then the median is

$$\begin{cases} a_{(n-1)/2} & \text{if } n \text{ is odd} \\ \frac{a_{n/2-1} + a_{n/2}}{2} & \text{if } n \text{ is even} \end{cases}$$

```
// Compute the median value of an input set of grades.
#include <algorithm>
#include <cmath>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <vector>

void read_scores(std::vector<int> & scores, std::ifstream & grade_str) {
  // scores can be changed in this function

  int x;  // input variable
  while (grade_str >> x) {
    scores.push_back(x);
  }
}

void compute_avg_and_std_dev(const std::vector<int>& s, double & avg, double & std_dev) {
  // s cannot be changed in this function

  // Compute and output the average value.
  int sum=0;
  for (unsigned int i = 0; i < s.size(); ++ i) {
    sum += s[i];
  }
  avg = double(sum) / s.size();

  // Compute the standard deviation
  double sum_sq = 0.0;
  for (unsigned int i=0; i < s.size(); ++i) {
    sum_sq += (s[i]-avg) * (s[i]-avg);
  }
  std_dev = sqrt(sum_sq / (s.size()-1));
}
```

```
double compute_median(const std::vector<int> & scores) {
  // Create a copy of the vector
  std::vector<int> scores_to_sort(scores);
  // Sort the values in the vector.  By default this is increasing order.
  std::sort(scores_to_sort.begin(), scores_to_sort.end());

  // Now, compute and output the median.
  unsigned int n = scores_to_sort.size();
  if (n%2 == 0)  // even number of scores
    return double(scores_to_sort[n/2] + scores_to_sort[n/2-1]) / 2.0;
  else
    return double(scores_to_sort[ n/2 ]);  // same as (n-1)/2 because n is odd
}

int main(int argc, char* argv[]) {
  if (argc != 2) {
    std::cerr << "Usage: " << argv[0] << " grades-file\n";
    return 1;
  }
  std::ifstream grades_str(argv[1]);
  if (!grades_str) {
    std::cerr << "Can not open the grades file " << argv[1] << "\n";
    return 1;
  }

  std::vector<int> scores;  // Vector to hold the input scores; initially empty.
  read_scores(scores, grades_str);   // Read the scores, as before

  // Quit with an error message if too few scores.
  if (scores.size() == 0) {
    std::cout << "No scores entered.  Please try again!" << std::endl;
    return 1;
  }

  // Compute the average, standard deviation and median
  double average, std_dev;
  compute_avg_and_std_dev(scores, average, std_dev);
  double median = compute_median(scores);

  // Output
  std::cout << "Among " << scores.size() << " grades: \n"
    << "  average = " << std::setprecision(3) << average << '\n'
    << "  std_dev = " << std_dev << '\n'
    << "   median = " << median << std::endl;
  return 0;
}
```

2.12 Passing Vectors (and Strings) As Parameters

The following outlines rules for passing vectors as parameters. The same rules apply to passing strings.

- If you are passing a vector as a parameter to a function and you want to make a (permanent) change to the vector, then you should pass it **by reference**.
  - This is illustrated by the function `read_scores` in the program `median_grade`.
  - This is very different from the behavior of arrays as parameters.
- What if you don't want to make changes to the vector or don't want these changes to be permanent?
  - The answer we've learned so far is to pass by value.
  - The problem is that the entire vector is copied when this happens! Depending on the size of the vector, this can be a considerable waste of memory.
- The solution is to pass by **constant reference**: pass it by reference, but make it a constant so that it can not be changed.

  - This is illustrated by the functions `compute_avg_and_std_dev` and `compute_median` in the program `median_grade`.

- As a general rule, you should not pass a container object, such as a vector or a string, by value because of the cost of copying.

CSCI-1200 Data Structures — Spring 2017
Lecture 3 — Classes I

Announcements

- Submitty team is working on an iClicker solution (we will put an announcement out on Piazza) when it's ready. This will let you register through Submitty instead of the iClicker site.
- Questions about Homework 1?

Today's Lecture

- Classes in C++ – Types and defining new types
- A `Date` class.
- Class declaration: member variables and member functions
- Using the class member functions
- Class scope
- Member function implementation
- Classes vs. structs
- Designing classes

Homework 1 Hints

- This section isn't in the printed lecture notes, but it is online.
- There are three major tasks in this assignment
  - Reading in the layout and commands
  - Managing the seats in a data structure
  - Managing the upgrade list (not to be confused with an STL `list` which we haven't yet covered)
- One of the problems is that many people naturally want to use `erase()`, but we haven't covered it
- More importantly, we haven't really discussed iterators, and they're very important to functions like `erase()`
- So how can we handle removing from a vector?
  - To empty out a vector, we can use `clear()`.
  - To remove the last value of a vector, we can use `pop_back()`
  - We could also remove an element by making a second vector that looks right, and then use an assignment `=`.
  - Let's look at a small program that exercises some of these concepts.

3.1 More Vector Sample Code

```
#include <iostream>
#include <vector>

void printVector(const std::vector<int>& vec, std::ostream& out){
    for(std::size_t i=0; i<vec.size(); i++){
        out << vec[i] << " ";
    }
    out << std::endl;
}

int main(){
    std::vector<int> a;
    std::vector<int> b;
    a.push_back(5);
    a.push_back(4);
    a.push_back(3);

    printVector(a, std::cout);
    printVector(b, std::cout);

    b = a;
    printVector(b, std::cout);

    b.pop_back();
    printVector(a, std::cout);
    a.clear();
    printVector(b, std::cout);
    printVector(a, std::cout);

    return 0;
}
```

3.2 Exercise

What will be the output of the "More Vector Sample Code" program above?

3.3 Types and Defining New Types

- What is a type? It is a structuring of memory plus a set of operations (functions) that can be applied to that structured memory.
  - Every C++ object has a type
  - The type tells us what the data means and what operations can be performed on the data
- Examples: integers, doubles, strings, and vectors.
- In many cases, when we are using a class we don't know how that memory is structured. Instead, what we really think about is the set of operations (functions) that can be applied.
- The basic ideas behind classes are **data abstraction** and **encapsulation**
  - Data abstraction hides details that don't matter from a certain point of view and identifies details that do matter.
  - The user sees only the interface to the object
  - The interface is the collection of data and operations that users of a class can access
    - For an int, you can access the value, perform addition etc.
    - For strings, concatenate, access characteres etc.

- Encapsulation is the packing of data and functions into a single component.
- Information hiding
  - Users have access to interface, but not implementation
  - No data item should be available any more than absolutely necessary
- To clarify, let's focus on strings and vectors. These are classes. We'll outline what we know about them:
  - The structure of memory within each class object
  - The set of operations defined
- We are now ready to start defining our own new types using classes.

3.4 Example: A Date Class

- Many programs require information about dates.
- Information stored about the date includes the month, the day and the year.
- Operations on the date include recording it, printing it, asking if two dates are equal, flipping over to the next day (incrementing), etc.

3.5 C++ Classes

- A C++ class consists of
  - a collection of member variables, usually `private`, and
  - a collection of member functions, usually `public`, which operate on these variables.
- `public` member functions can be accessed directly from outside the class,
- `private` member functions and member variables can only be accessed indirectly from outside the class, through public member functions.
- We will look at the example of the `Date` class declaration.

3.6 Using C++ classes

- We have been using C++ classes (from the standard library) already this semester, so studying how the `Date` class is used is straightforward:

```
// Program:  date_main.cpp
// Purpose:  Demonstrate use of the Date class.

#include <iostream>
#include "date.h"

int main() {
  std::cout << "Please enter today's date.\n"
    << "Provide the  month, day and year: ";
  int month, day, year;
  std::cin >> month >> day >> year;
  Date today(month, day, year);

  Date tomorrow(today.getMonth(), today.getDay(), today.getYear());
  tomorrow.increment();

  std::cout << "Tomorow is ";
  tomorrow.print();
  std::cout << std::endl;

  Date Sallys_Birthday(2,3,1995);
  if (sameDay(tomorrow, Sallys_Birthday)) {
    std::cout << "Hey, tomorrow is Sally's birthday!\n";
  }

  std::cout << "The last day in this month is " << today.lastDayInMonth() << std::endl;
  return 0;
}
```

- **Important:** Each object we create of type `Date` has its own distinct member variables.
- Calling class member functions for class objects uses the "dot" notation. For example, `tomorrow.increment();`
- Note: We don't need to know the implementation details of the class member functions in order to understand this example. This is an important feature of object oriented programming and class design.

3.7 Exercise

Add code to `date_main.cpp` to read in another date, check if it is a leap-year, and check if it is equal to `tomorrow`. Output appropriate messages based on the results of the checks.

3.8 Class Declaration (`date.h`) & Implementation (`date.cpp`)

A class implementation usually consists of 2 files. First we'll look at the *header file* `date.h`

```
// File:     date.h
// Purpose:  Header file with declaration of the Date class, including
//   member functions and private member variables.

class Date {
public:
  Date();
  Date(int aMonth, int aDay, int aYear);

  // ACCESSORS
  int getDay() const;
  int getMonth() const;
  int getYear() const;

  // MODIFIERS
  void setDay(int aDay);
  void setMonth(int aMonth);
  void setYear(int aYear);
  void increment();

  // other member functions that operate on date objects
  bool isEqual(const Date& date2) const;  // same day, month, & year?
  bool isLeapYear() const;
  int lastDayInMonth() const;
  bool isLastDayInMonth() const;
  void print() const;                     // output as month/day/year

private:  // REPRESENTATION (member variables)
  int day;
  int month;
  int year;
};

// prototypes for other functions that operate on class objects are often
// included in the header file, but outside of the class declaration
bool sameDay(const Date &date1, const Date &date2); // same day & month?
```

And here is the other part of the class implementation, the *implementation file* `date.cpp`

```
// File:    date.cpp
```

```
// Purpose: Implementation file for the Date class.

#include <iostream>
#include "date.h"

// array to figure out the number of days, it's used by the auxiliary function daysInMonth
const int DaysInMonth[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

Date::Date() { //default constructor
  day = 1;
  month = 1;
  year = 1900;
}

Date::Date(int aMonth, int aDay, int aYear) { // construct from month, day, & year
  month = aMonth;
  day = aDay;
  year = aYear;
}

int Date::getDay() const {
  return day;
}

int Date::getMonth() const {
  return month;
}

int Date::getYear() const {
  return year;
}

void Date::setDay(int d) {
  day = d;
}

void Date::setMonth(int m) {
  month = m;
}

void Date::setYear(int y) {
  year = y;
}

void Date::increment() {
  if (!isLastDayInMonth()) {
    day++;
  } else {
    day = 1;
    if (month == 12) {  // December
      month = 1;
      year++;
    } else {
      month++;
    }
  }
}

bool Date::isEqual(const Date& date2) const {
  return day == date2.day && month == date2.month && year == date2.year;
}

bool Date::isLeapYear() const {
  return (year%4 ==0 && year % 100 != 0) || year%400 == 0;
}
```

```
int Date::lastDayInMonth() const {
  if (month == 2 && isLeapYear())
    return 29;
  else
    return DaysInMonth[ month ];
}

bool Date::isLastDayInMonth() const {
  return day == lastDayInMonth();   // uses member function
}

void Date::print() const {
  std::cout << month << "/" << day << "/" << year;
}

bool sameDay(const Date& date1, const Date& date2) {
  return date1.getDay() == date2.getDay() && date1.getMonth() == date2.getMonth();
}
```

3.9 Class scope notation

- `Date::` indicates that what follows is within the scope of the class.
- Within class scope, the member functions and member variables are accessible without the name of the object.

3.10 Constructors

These are special functions that initialize the values of the member variables. You have already used constructors for string and vector objects.

- The syntax of the call to the constructor mixes variable definitions and function calls. (See `date_main.cpp`)
- "Default constructors" have no arguments.
- Multiple constructors are allowed, just like multiple functions with the same name are allowed. The compiler determines which one to call based on the types of the arguments (just like any other function call).
- When a new object is created, *EXACTLY one constructor for the object is called.*

3.11 Member Functions

Member functions are like ordinary functions except:

- They can access and modify the object's member variables.
- They can call the other member functions without using an object name.
- Their syntax is slightly different because they are defined within class scope.

For the `Date` class:

- The `set` and `get` functions access and change a day, month or year.
- The `increment` member function uses another member function, `isLastDayInMonth`.
- `isEqual` accepts a second `Date` object and then accesses its values directly using the dot notation. Since we are inside class `Date` scope, this is allowed. The name of the second object, `date2`, is required to indicate that we are interested in its member variables.
- `lastDayInMonth` uses the const array defined at the start of the .cpp file.

More on member functions:

- When the member variables are *private*, the only means of accessing them and changing them from outside the class is through member functions.
- If member variables are made *public*, they can be accessed directly. This is usually considered bad style and should not be used in this course.

- Functions that are not members of the `Date` class must interact with `Date` objects through the class public members (a.k.a., the "public interface" declared for the class). One example is the function `sameDay` which accepts two `Date` objects and compares them by accessing their day and month values through their public member functions.

3.12 Header Files (.h) and Implementation Files (.cpp)

The code for the Date example is in three files:

- The *header file*, `date.h`, contains the class declaration.
- The *implementation file*, `date.cpp`, contains the member function definitions. Note that `date.h` is `#include`'ed.
- `date_main.cpp` contains the code outside the class. Again `date.h` again is `#include`'ed.
- The files `date.cpp` and `date_main.cpp` are compiled separately and then linked to form the executable program.
  - `g++ -c -Wall date.cpp`
  - `g++ -c -Wall date_main.cpp`
  - `g++ -o date.exe date.o date_main.o`
  - or all on one line `g++ -o date.exe date.cpp date_main.cpp`
- Different organizations of the code are possible, but not preferable. In fact, we could have put all of the code from the 3 files into a single file `main.cpp`. In this case, we would not have to compile two separate files.
- In many large projects, programmers establish follow a convention with two files per class, one header file and one implementation file. This makes the code more manageable and is recommended in this course.

3.13 Constant member functions

Member functions that do not change the member variables should be declared `const`

- For example: `bool Date::isEqual(const Date &date2) const;`
- This must appear consistently in **both** the member function declaration in the class declaration (in the `.h` file) and in the member function definition (in the `.cpp` file).
- `const` objects (usually passed into a function as parameters) can ONLY use const member functions. *Remember, you should only pass objects by value under special circumstances. In general, pass all objects by reference so they aren't copied, and by const reference if you don't want/need them to change.*
- While you are learning, you will probably make mistakes in determining which member functions should or should not be const. Be prepared for compile warnings & errors, and read them carefully.

3.14 Exercise

Add a member function to the `Date` class to add a given number of days to the Date object. The number should be the only argument and it should be an unsigned int. Should this function be `const`?

3.15 Classes vs. structs

- Technically, a `struct` is a `class` where the default protection is `public`, not `private`.
  - As mentioned above, when a member variable is `public` it can be accessed and changed directly using the dot notation: `tomorrow.day = 52;` We can see immediately why this is dangerous (and an example of bad programming style) because a day of 52 is invalid!
- The usual practice of using `struct` is all public members and no member functions.

Rule for the duration of the Data Structures course: You may not declare new struct types, and class member variables should not be made public. This rule will ensure you get plenty of practice writing C++ classes with good programming style.

3.16 C++ vs. Java Classes

- In C++, classes have sections labeled `public` and `private`, but there can be multiple public and private sections. In Java, each individual item is tagged public or private.
- Class declarations and class definitions are separated in C++, whereas they are together in Java.
- In C++ there is a semi-colon at the very end of the class declaration (after the }).

3.17 C++ vs. Python Classes

- Python classes have a single constructor, __init__.
- Python is dynmaically typed. Class attributes such as members are defined by assignment.
- Python classes do not have private members. Privacy is enforced by convention.
- Python methods have an explicit *self* reference variable.

3.18 Designing and implementing classes

This takes a lot of practice, but here are some ideas to start from:

- Begin by outlining what the class objects should be able to do. This gives a start on the member functions.
- Outline what data each object keeps track of, and what member variables are needed for this storage.
- Write a draft class declaration in a `.h` file.
- Write code that uses the member functions (e.g., the `main` function). Revise the class `.h` file as necessary.
- Write the class `.cpp` file that implements the member functions.

In general, don't be afraid of major rewrites if you find a class isn't working correctly or isn't as easy to use as you intended. This happens frequently in practice!

CSCI-1200 Data Structures — Spring 2017
Lecture 4 — Classes II: Sort, Non-member Operators

Announcements

- Excercise solutions will be posted to the calendar.
- Submitty iClicker registration is still open. **Even if you already registered on the iClicker website**, submit your code on Submitty.
- Starting with HW2, when Submitty opens for the homework assignment, there may be a message at the top regarding an extra late day for earning enough autograder points by Wednesday night.
- Practice problems for Exam 1 will be posted Monday, but the solutions will not be posted until the weekend.
- We will talk more about the exam next Tuesday.

Review from Lecture 3

- C++ classes, member variables and member functions, class scope, public and private
- Nuances to remember
  - Within class scope (within the code of a member function) member variables and member functions of that class may be accessed without providing the name of the class object.
  - Within a member function, when an object of the same class type has been passed as an argument, direct access to the private member variables of that object is allowed (using the '.' notation).
- Classes vs. structs
- Designing classes
- Common error

Today's Lecture

- Extended example of student grading program
- Passing comparison functions to `sort`
- Non-member operators

4.1 Example: Student Grades

Our goal is to write a program that calculates the grades for students in a class and outputs the students and their averages in alphabetical order. The program source code is broken into three parts:

- Re-use of statistics code from Lecture 2.
- Class `Student` to record information about each student, including name and grades, and to calculate averages.
- The main function controls the overall flow, including input, calculation of grades, and output.

```
// File:     main_student.cpp
// Purpose:  Compute student averages and output them alphabetically.
#include <algorithm>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <vector>
#include "student.h"

int main(int argc, char* argv[]) {
  if (argc != 3) {
    std::cerr << "Usage:\n  " << argv[0] << " infile-students outfile-grades\n";
    return 1;
  }
  std::ifstream in_str(argv[1]);
  if (!in_str) {
```

```
    std::cerr << "Could not open " << argv[1] << " to read\n";
    return 1;
  }
  std::ofstream out_str(argv[2]);
  if (!out_str) {
    std::cerr << "Could not open " << argv[2] << " to write\n";
    return 1;
  }

  int num_homeworks, num_tests;
  double hw_weight;
  in_str >> num_homeworks >> num_tests >> hw_weight;
  std::vector<Student> students;
  Student one_student;

  // Read the students, one at a time.
  while(one_student.read(in_str, num_homeworks, num_tests)) {
    students.push_back(one_student);
  }

  // Compute the averages.  At the same time, determine the maximum name length.
  unsigned int i;
  unsigned int max_length = 0;
  for (i=0; i<students.size(); ++i) {
    students[i].compute_averages(hw_weight);
    unsigned int tmp_length;
    tmp_length = students[i].first_name().size() + students[i].last_name().size();
    max_length = std::max(max_length, tmp_length);
  }
  max_length += 2;   // account for the output padding with ", "

  // Sort the students alphabetically by name.
  std::sort(students.begin(), students.end(), less_names);

  // Output a header...
  out_str << "\nHere are the student semester averages\n";
  const std::string header = "Name" + std::string(max_length-4, ' ') + "  HW  Test Final";
  const std::string underline(header.size(), '-');
  out_str << header << '\n' << underline << std::endl;

  // Output the students...
  for (i=0; i<students.size(); ++i) {
    unsigned int length = students[i].last_name().size() +
      students[i].first_name().size() + 2;
    students[i].output_name(out_str);
    out_str << std::string(max_length - length, ' ') << "  ";
    students[i].output_averages(out_str);
  }

  return 0;  // everything fine
}
```

4.2 Declaration of Class `Student`

- Stores names, id numbers, scores and averages. The scores are stored using a vector! Member variables of a class can be other classes!
- Functionality is relatively simple: input, compute average, provide access to names and averages, and output.
- No constructor is explicitly provided: `Student` objects are built through the `read` function.
  (Other code organization/designs are possible!)
- Overall, the `Student` class design differs substantially in style from the `Date` class design. We will continue to see different styles of class designs throughout the semester.
- Note the helpful convention used in this example: all member variable names end with the "`_`" character.
- The special pre-processor directives `#ifndef __student_h_`, `#define __student_h_`, and `#endif` ensure that this files is included at most once per `.cpp` file.

  For larger programs with multiple class files and interdependencies, these lines are essential for successful compilation. We suggest you get in the habit of adding these *include guards* to all your header files.

```
// File:     student.h
// Purpose:  Header for declaration of student record class and associated functions.

#ifndef __student_h_
#define __student_h_

#include <iostream>
#include <string>
#include <vector>

class Student {
public:
  // ACCESSORS
  const std::string& first_name() const { return first_name_; }
  const std::string& last_name() const { return last_name_; }
  const std::string& id_number() const { return id_number_; }
  double hw_avg() const { return hw_avg_; }
  double test_avg() const { return test_avg_; }
  double final_avg() const { return final_avg_; }

  bool read(std::istream& in_str, unsigned int num_homeworks, unsigned int num_tests);
  void compute_averages(double hw_weight);
  std::ostream& output_name(std::ostream& out_str) const;
  std::ostream& output_averages(std::ostream& out_str) const;

private: // REPRESENTATION
  std::string first_name_;
  std::string last_name_;
  std::string id_number_;
  std::vector<int> hw_scores_;
  double hw_avg_;
  std::vector<int> test_scores_;
  double test_avg_;
  double final_avg_;
};

bool less_names(const Student& stu1, const Student& stu2);
#endif
```

4.3 Automatic Creation of Two Constructors By the Compiler

- Two constructors are created automatically by the compiler because they are needed and used.
- The first is a default constructor which has no arguments and *just calls the default constructor for each of the member variables.* The prototype is `Student();`

  The default constructor is called when the `main()` function line `Student one_student;` is executed.

  If you wish a different behavior for the default constructor, you must declare it in the `.h` file and provide the alternate implementation.
- The second automatically-created constructor is a "copy constructor", whose only argument is a `const` reference to a `Student` object. The prototype is `Student(const Student &s);`

  This constructor *calls the copy constructor for each member variable* to copy the member variables from the passed `Student` object to the corresponding member variables of the `Student` object being created. If you wish a different behavior for the copy constructor, you must declare it an provide the alternate implementation.

  The copy constructor is called during the vector `push_back` function in copying the contents of `one_student` to a new `Student` object on the back of the vector `students`.
- The behavior of automatically-created default and copy constructors is often, but not always, what's desired. When they do what's desired, the convention is to not write them explicitly.
- Later in the semester we will see circumstances where writing our own default and copy constructors is crucial.

4.4 Implementation of Class `Student`

- The `read` function is fairly sophisticated and depends heavily on the expected structure of the input data. It also has a lot of error checking.
  - In many class designs, this type of input would be done by functions outside the class, with the results passed into a constructor. Generally prefer this style because it separates elegant class design from clunky I/O details.
- The accessor functions for the names are defined within the class declaration in the header file. **In this course, you are allowed to do this for one-line functions only!** For complex classes, including long definitions within the header file has dependency and performance implications.
- The computation of the averages uses some but not all of the functionality from `stats.h` and `stats.cpp` (not included in your handout).
- Output is split across two functions. Again, stylistically, it is sometimes preferable to do this outside the class.

```
// File:     student.cpp
// Purpose:  Implementation of the class Student

#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include "student.h"
#include "std_dev.h"

// Read information about a student, returning true if the information was read correctly.
bool Student::read(std::istream& in_str, unsigned int num_homeworks, unsigned int num_tests) {

  // If we don't find an id, we've reached the end of the file & silently return false.
  if (!(in_str >> id_number_)) return false;
  // Once we have an id number, any other failure in reading is treated as an error.

  // read the name
  if (! (in_str >> first_name_ >> last_name_)) {
    std::cerr << "Failed reading name for student " << id_number_ << std::endl;
    return false;
  }

  unsigned int i;
  int score;

  // Read the homework scores
  hw_scores_.clear();
  for (i=0; i<num_homeworks && (in_str >> score); ++i)
    hw_scores_.push_back(score);
  if (hw_scores_.size() != num_homeworks) {
    std::cerr << "Pre-mature end of file or invalid input reading "
              << "hw scores for " << id_number_ << std::endl;
    return false;
  }

  // Read the test scores
  test_scores_.clear();
  for (i=0; i<num_tests && (in_str >> score); ++i)
    test_scores_.push_back(score);
  if (test_scores_.size() != num_tests) {
    std::cerr << "Pre-mature end of file or invalid input reading "
              << "test scores for" << id_number_ << std::endl;
    return false;
  }
  return true;  // everything was fine
}

// Compute and store the hw, test and final average for the student.
void Student::compute_averages(double hw_weight) {
  double dummy_stddev;
  avg_and_std_dev(hw_scores_, hw_avg_, dummy_stddev);
  avg_and_std_dev(test_scores_, test_avg_, dummy_stddev);
  final_avg_ = hw_weight * hw_avg_ + (1 - hw_weight) * test_avg_;
}
```

```
std::ostream& Student::output_name(std::ostream& out_str) const {
  out_str << last_name_ << ", " << first_name_;
  return out_str;
}

std::ostream& Student::output_averages(std::ostream& out_str) const {
  out_str << std::fixed << std::setprecision(1);
  out_str << hw_avg_ << " " << test_avg_ << " " << final_avg_ << std::endl;
  return out_str;
}


// Boolean function to define alphabetical ordering of names.  The vector sort
// function requires that the objects be passed by CONSTANT REFERENCE.
bool less_names(const Student& stu1, const Student& stu2) {
  return stu1.last_name() < stu2.last_name() ||
         (stu1.last_name() == stu2.last_name() && stu1.first_name() < stu2.first_name());
}

/* alternative version
bool less_names(const Student& stu1, const Student& stu2) {
  if (stu1.last_name() < stu2.last_name())
    return true;
  else if (stu1.last_name() == stu2.last_name())
    return stu1.first_name() < stu2.first_name();
  else
    return false;
}

*/
```

4.5 Exercise

Add code to the end of the `main()` function to compute and output the average of the semester grades and to output a list of the semester grades sorted into increasing order.

4.6 Providing Comparison Functions to Sort

Consider sorting the students vector:

- If we used `sort(students.begin(), students.end());` the sort function would try to use the `<` operator on `student` objects to sort the students, just as it earlier used the `<` operator on doubles to sort the grades. However, this doesn't work because there is no such operator on `Student` objects.
- Fortunately, the sort function can be called with a third argument, a comparison function:
  `sort(students.begin(), students.end(), less_names);`

  `less_names`, defined in `student.cpp`, is a function that takes two const references to `Student` objects and returns true if and only if the first argument should be considered "less" than the second in the sorted order. `less_names` uses the `<` operator defined on `string` objects to determine its ordering.

4.7 Exercise

Write a function `greater_averages` that could be used in place of `less_names` to sort the `students` vector so that the student with the highest semester average is first.

4.8 Operators As Non-Member Functions

- A second option for sorting is to define a function that creates a `<` operator for `Student` objects! At first, this seems a bit weird, but it is extremely useful.
- Let's start with syntax. The expressions `a < b` and `x + y` are really function calls! Syntactically, they are equivalent to `operator< (a, b)` and `operator+ (x, y)` respectively.
- When we want to write our own operators, we write them as functions with these weird names.
- For example, if we write:

```
bool operator< (const Student& stu1, const Student& stu2) {
  return  stu1.last_name() < stu2.last_name() ||
          (stu1.last_name() == stu2.last_name() &&
           stu1.first_name() < stu2.first_name());
}
```

  then the statement `sort(students.begin(), students.end());` will sort `Student` objects into alphabetical order.
- Really, the only weird thing about operators is their syntax.
- We will have many opportunities to write operators throughout this course. Sometimes these will be made class member functions, but more on this in a later lecture.

4.9 A Word of Caution about Operators

- Operators should only be defined if their meaning is intuitively clear.
- `operator<` on `Student` objects fails the test because the natural ordering on these objects is not clear.
- By contrast, `operator<` on `Date` objects is much more natural and clear.

4.10 Exercise

Write an `operator<` for comparing two `Date` objects.

4.11 Another Class Example: Alphabetizing Names

```
// name_main.cpp
// Demonstrates another example with the use of classes, including an output stream operator
#include <algorithm>
#include <iostream>
#include <vector>
#include "name.h"

int main() {
  std::vector<Name> names;
  std::string first, last;
  std::cout <<"\nEnter a sequence of names (first and last) and this program will alphabetize them\n";

  while (std::cin >> first >> last) {
    names.push_back(Name(first, last)); }

  std::sort(names.begin(), names.end());
  std::cout << "\nHere are the names, in alphabetical order.\n";

  for (int i = 0; i < names.size(); ++i) {
    std::cout << names[i] << "\n"; }

  return 0;
}
```

4.12 Name Class Declaration & Implementation

```
#ifndef __NAME__
#define __NAME__

// name.h
#include <iostream>
#include <string>

class Name {
public:

  // CONSTRUCTOR
  Name(const std::string& fst, const std::string& lst);

  // ACCESSORS
  // Providing a const reference to the string allows the string to be
  // examined and treated as an r-value without the cost of copying it.
  const std::string& first() const { return first_; }
  const std::string& last() const { return last_; }

  // MODIFIERS
  void set_first(const std::string & fst) { first_ = fst; }
  void set_last(const std::string& lst) { last_ = lst; }

private: // REPRESENTATION
  std::string first_, last_;
};

// operator< to allow sorting
bool operator< (const Name& left, const Name& right);

// operator<< to allow output
std::ostream& operator<< (std::ostream& ostr, const Name& n);

#endif
```

```
// name.cpp
#include "name.h"

// Here we use special syntax to call the string class copy constructors
Name::Name(const std::string& fst, const std::string& lst)
  : first_(fst), last_(lst)
{}

// The alternative implementation below first calls the default string
// constructor for the two variables, then performs an assignment in
// the body of the constructor function.
/*
Name::Name(const std::string& fst, const std::string& lst) {
  first_ = fst;
  last_ = lst;
}
*/

// operator<
bool operator< (const Name& left, const Name& right) {
  return left.last()<right.last() ||
    (left.last()==right.last() && left.first()<right.first());
}

// operator<< is the output stream operator.  It takes two arguments:
// the stream (such as cout) and the object to be output.  The <<
// operator should always return a reference to the output stream to
// allow a sequence of outputs in a single statement.
std::ostream& operator<< (std::ostream& ostr, const Name& n) {
  ostr << n.first() << " " << n.last();
  return ostr;
}
```

CSCI-1200 Data Structures — Spring 2017
Lecture 5 — Pointers, Arrays, Pointer Arithmetic

Announcements

- Submitty iClicker registration is still open. **Even if you already registered on the iClicker website**, submit your code on Submitty.
- Starting with HW2, when Submitty opens for the homework assignment, there may be a message at the top regarding an extra late day for earning enough autograder points by Wednesday night.
- In fact, right now it's set for 12 autograder points. This is the number you see and is the points from visible test cases.

Announcements: Test 1 Information

- Test 1 will be held **Monday, Feb 6th, 2017 from 6-7:50pm**,
- Your seating assignment will be posted on Submitty / through the gradesheet. Details will be given out Friday. No make-ups will be given except for pre-approved absence or illness, and a written excuse from the Dean of Students or the Student Experience office or the RPI Health Center will be required.

  Contact Mrs. Eberwein by email by Friday Feb 3rd to arrange for extra time accommodations. You can alternately e-mail the ds_instructors list.
- Coverage: Lectures 1-6, Labs 1-3, and Homeworks 1-2.
- Closed-book and closed-notes *except for 1 sheet of notes on 8.5x11 inch paper (front & back) that may be handwritten or printed*. Computers, cell-phones, calculators, PDAs, music players, etc. are not permitted and must be turned off and placed under your desk.
- All students must bring their Rensselaer photo ID card.
- At the start of the exam, proctors will check that you have your ID card, and if you have a sheet of notes, they will staple it to the back of your exam.
- Practice problems from previous exams are available on the course website. Solutions to the problems will be posted on Sunday. The best way to prepare is to completely work through and write out your solution to each problem, *before* looking at the answers.
- The exam *will* involve handwriting code on paper (and other short answer problem solving). Neat legible handwriting is appreciated. We will somewhat forgiving about minor syntax errors – it will be graded by humans not computers :)

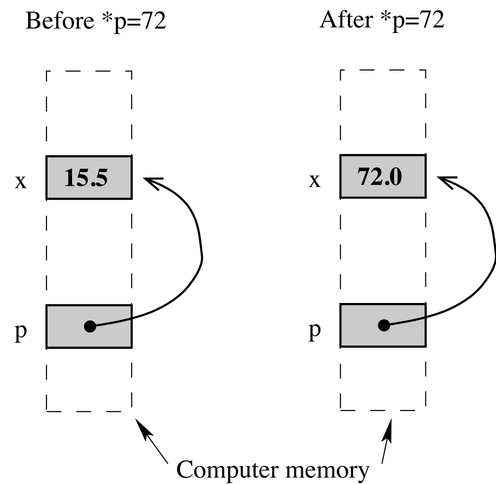
Review from Last Week

- C++ class syntax, designing classes, classes vs. structs;
- Passing comparison functions to `sort`; Non-member operators.
- More practice with `const` and reference (the '`&`')

Today's Lecture — Pointers and Arrays

- Pointers store memory addresses.
- They can be used to access the values stored at their stored memory address.
- They can be incremented, decremented, added and subtracted.
- Dynamic memory is accessed through pointers.
- Pointers are also the primitive mechanism underlying vector iterators, which we have used with `std::sort` and will use more extensively throughout the semester.

5.1 Pointer Example



- Consider the following code segment:

```
float x = 15.5;
float *p; /* equiv:  float* p;  or  float * p;  */
p = &x;
*p = 72;
if ( x > 20 )
  cout << "Bigger\n";
else
  cout << "Smaller\n";
```

The output is `Bigger`

because `x == 72.0`. What's going on?

5.2 Pointer Variables and Memory Access

- `x` is an ordinary float, but `p` is a pointer that can hold the memory address of a float variable. The difference is explained in the picture above.
- Every variable is attached to a location in memory. This is where the value of that variable is stored. Hence, we draw a picture with the variable name next to a box that represents the memory location.
- Each memory location also has an address, which is itself just an index into the giant array that is the computer memory.
- The value stored in a pointer variable is an address in memory. The statement `p = &x;` takes the address of `x`'s memory location and stores it (the address) in the memory location associated with `p`.
- Since the value of this address is much less important than the fact that the address is `x`'s memory location, we depict the address with an arrow.
- The statement: `*p = 72;` causes the computer to get the memory location stored at `p`, then go to that memory location, and store 72 there. This writes the 72 in `x`'s location.

  Note: `*p` is an *l-value* in the above expression.

5.3 Defining Pointer Variables

- In the example below, `p`, `s` and `t` are all pointer variables (pointers, for short), but `q` is NOT. You need the * before each variable name.

```
int * p, q;
float *s, *t;
```

- There is no initialization of pointer variables in this two-line sequence, so the statement below is dangerous, and may cause your program to crash! (It won't crash if the uninitialized value happens to be a legal address.)

```
*p = 15;
```

5.4 Operations on Pointers

- The unary (single argument/operand) operator * in the expression `*p` is the "dereferencing operator". It means "follow the pointer" `*p` can be either an l-value or an r-value, depending on which side of the `=` it appears on.
- The unary operator & in the expression `&x` means "take the memory address of."
- Pointers can be assigned. This just copies memory addresses as though they were values (which they are). Let's work through the example below (and draw a picture!). What are the values of `x` and `y` at the end?

```
float x=5, y=9;
float *p = &x, *q = &y;
*p = 17.0;
*q = *p;
q = p;
*q = 13.0;
```

- Assignments of integers or floats to pointers and assignments mixing pointers of different types are illegal. Continuing with the above example:

```
int *r;
r = q;      //  Illegal: different pointer types;
p = 35.1;   //  Illegal: float assigned to a pointer
```

- Comparisons between pointers of the form `if ( p == q )` or `if ( p != q )` are legal and very useful! Less than and greater than comparisons are also allowed. These are useful only when the pointers are to locations within an array.

5.5 Exercise

- Draw a picture for the following code sequence. What is the output to the screen?

```
int x = 10, y = 15;
int *a = &x;
cout << x << " " << y << endl;
int *b = &y;
*a = x * *b;
cout << x << " " << y << endl;
int *c = b;
*c = 25;
cout << x << " " << y << endl;
```

5.6 Null Pointers

- Like the `int` type, pointers are not default initialized. We should assume it's a garbage value, leftover from the previous user of that memory.
- Pointers that don't (yet) point anywhere useful should be explicitly assigned to `NULL`.
  - `NULL` is equal to the integer 0, which is a legal pointer value (you can store the `NULL` in a pointer variable).
  - But `NULL` is not a valid memory location you are allowed to read or write. If you try to dereference or *follow a* `NULL` *pointer*, your program will immediately crash. You may see a segmentation fault, a bus error, or something about a null pointer dereference.
  - *NOTE:* In C++11 (the server still uses C++03), we are encouraged to switch to use `nullptr`, to avoid some subtle situations where `NULL` is incorrectly seen as an `int` type instead of a pointer.
  - We indicate a `NULL` value in diagrams with a slash through the memory location box.
- Comparing a pointer to NULL is very useful. It can be used to indicate whether or not a pointer variable is pointing at a useable memory location. For example,

```
if ( p != NULL )
  cout << *p << endl.
```

  tests to see if `p` is pointing somewhere that appears to be useful before accessing and printing the value stored at that location.

- But don't make the mistake of assuming pointers are automatically initialized to NULL.

5.7 Arrays

- Here's a quick example to remind you about how to use an array:

```
const int n = 10;
double a[n];
int i;
for ( i=0; i<n; ++i )
    a[i] = sqrt( double(i) );
```

- Remember: the size of array `a` is fixed at compile time. STL vectors act like arrays, but they can grow and shrink dynamically in response to the demands of the application.

5.8 Stepping through Arrays with Pointers (Array *Iterators*)

- The array code above that uses `[]` *subscripting*, can be equivalently rewritten to use pointers:

```
const int n = 10;
double a[n];
double *p;
for ( p=a; p<a+n; ++p )
   *p = sqrt( p-a );
```

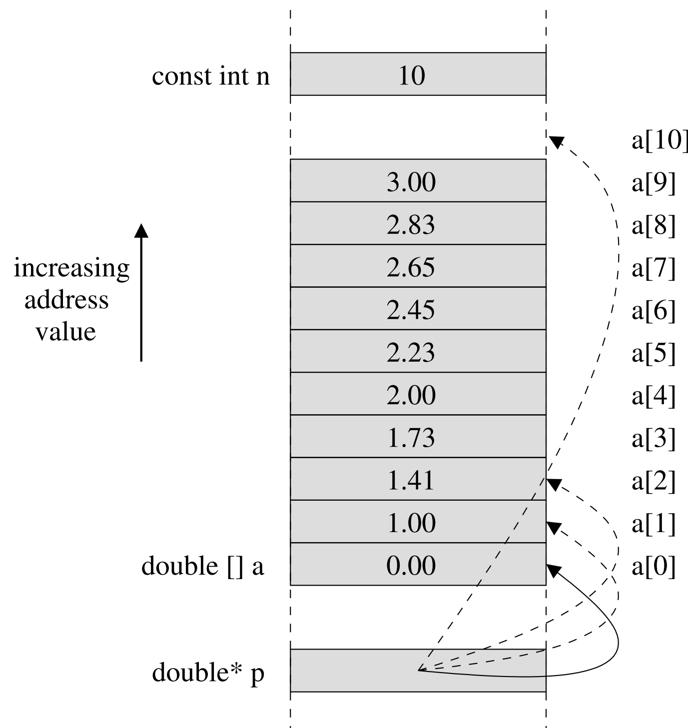
- The assignment: `p = a;` takes the address of the start of the array and assigns it to `p`.
- This illustrates the important fact that the name of an array is in fact **a pointer to the start of a block of memory**. We will come back to this several times! We could also write this line as: `p = &a[0];` which means "find the location of `a[0]` and take its address".
- By incrementing, `++p`, we make `p` point to the next location in the array.
  - When we increment a pointer we don't just add one byte to the address, we add the number of bytes (*sizeof*) used to store one object of the specific type of that pointer. Similarly, basic addition/subtraction of pointer variables is done in multiples of the *sizeof* the type of the pointer.
  - Since the type of `p` is `double`, and the size of `double` is 8 bytes, we are actually adding 8 bytes to the address when we execute `++p`.
- The test `p<a+n` checks to see if the value of the pointer (the address) is less than n array locations beyond the start of the array.

  In this example, `a+n` is the memory location 80 bytes after the start of the array (n = 10 slots * 8 bytes per slot).

  We could equivalently have used the test `p != a+n`
- In the assignment:

```
*p = sqrt( p-a )
```

  `p-a` is the number of array locations (multiples of 8 bytes) between p and the start. This is an integer. The square root of this value is assigned to `*p`.
- Here's a picture to explain this example:



- Note that there may or may not be unused memory between your array and the other local variables. Similarly, the order that your local variables appear on the stack is not guaranteed (the compiler may rearrange things a bit in an attempt to optimize performance or memory usage). A *buffer overflow* (attempting to access an illegal array index) may or may not cause an immediate failure – depending on the layout of other critical program memory.

5.9 Sorting an Array

- Arrays may be sorted using `std::sort`, just like vectors. Pointers are used in place of iterators. For example, if `a` is an array of doubles and there are `n` values in the array, then here's how to sort the values in the array into increasing order:

```
std::sort( a, a+n );
```

5.10 Exercises

For each of the following problems, you may only use pointers and not subscripting:

1. Write code to print the array `a` backwards, using pointers.

2. Write code to print every other value of the array `a`, again using pointers.

3. Write a function that checks whether the contents of an array of doubles are sorted into increasing order. The function must accept two arguments: a pointer (to the start of the array), and an integer indicating the size of the array.

5.11 C Calling Convention

- We take for granted the non-trivial task of passing data to a helper function, getting data back from that function, and seamlessly continuing on with the program. *How does that work??*
- A *calling convention* is a standardized method for passing arguments between the caller and the function. Calling conventions vary between programming languages, compilers, and computer hardware.
- In C on x86 architectures here is a generalization of what happens:
  1. The caller puts all the arguments on the *stack*, in reverse order.
  2. The caller puts the address of its code on the stack (the *return address*).
  3. Control is transferred to the callee.
  4. The callee puts any local variables on the stack.
  5. The callee does its work and puts the return value in a special *register* (storage location).
  6. The callee removes its local variables from the stack.
  7. Control is transferred by removing the address of the caller from the stack and going there.
  8. The caller removes the arguments from the stack.
- On x86 architectures the addresses on the stack are in descending order. This is not true of all hardware.

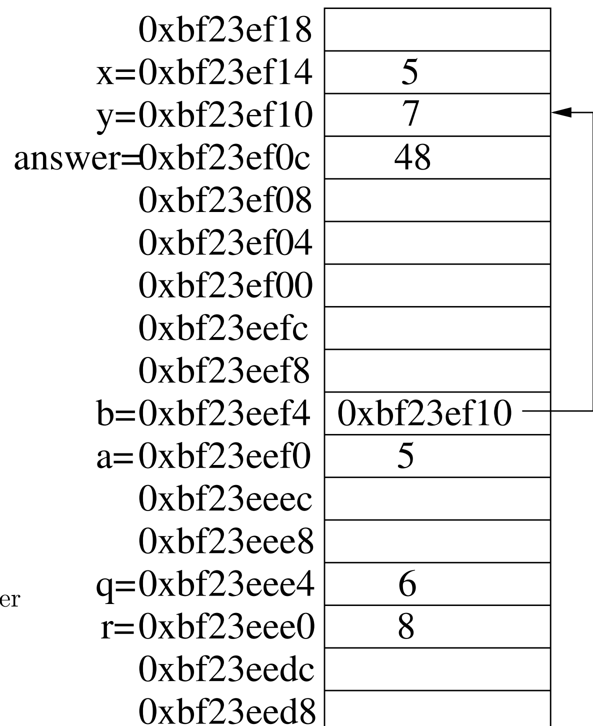
5.12 Poking around in the Stack & Looking for the C Calling Convention

- Let's look more closely at an example of where the compiler stores our data. Specifically, let's print out the addresses and values of the local variables and function parameters:

```
int foo(int a, int *b) {
  int q = a+1;
  int r = *b+1;
  std::cout << "address of a = " << &a << std::endl;
  std::cout << "address of b = " << &b << std::endl;
  std::cout << "address of q = " << &q << std::endl;
  std::cout << "address of r = " << &r << std::endl;
  std::cout << "value at " << &a << " = " << a << std::endl;
  std::cout << "value at " << &b << " = " << b << std::endl;
  std::cout << "value at " << b << " = " << *b << std::endl;
  std::cout << "value at " << &q << " = " << q << std::endl;
  std::cout << "value at " << &r << " = " << r << std::endl;
  return q*r;
}

int main() {
  int x = 5;
  int y = 7;
  int answer = foo (x, &y);
  std::cout << "address of x = " << &x << std::endl;
  std::cout << "address of y = " << &y << std::endl;
  std::cout << "address of answer = " << &answer << std::endl;
  std::cout << "value at " << &x << " = " << x << std::endl;
  std::cout << "value at " << &y << " = " << y << std::endl;
  std::cout << "value at " << &answer << " = " << answer << std::endl;
}
```

- Note that the first function parameters is regular integer, passed by copy. The second parameter is a passed in as a pointer.
- Note that we can print out data values or pointers – the address is printed as a big integer in hexadecimal format (beginning with "0x"). This example was compiled as 32-bit program, so our addresses are 32-bits. A 64-bit program will have longer addresses.
- Let's look at the program output and reverse engineer a drawing of the stack:

```
address of a = 0xbf23eef0
address of b = 0xbf23eef4
address of q = 0xbf23eee4
address of r = 0xbf23eee0
value at 0xbf23eef0 = 5
value at 0xbf23eef4 = 0xbf23ef10
value at 0xbf23ef10 = 7
value at 0xbf23eee4 = 6
value at 0xbf23eee0 = 8
address of x = 0xbf23ef14
address of y = 0xbf23ef10
address of answer = 0xbf23ef0c
value at 0xbf23ef14 = 5
value at 0xbf23ef10 = 7
value at 0xbf23ef0c = 48
```



- Note: The unlabeled portions in our diagram of the stack will include the frame pointer, the return address, temp variables (complex C++ expressions turn into many smaller steps of assembly), space to save registers, and padding between variables to meet alignment requirements.
- Note: Different compilers and/or different optimization levels will produce a different stack diagram.

CSCI-1200 Data Structures — Spring 2017
Lecture 6 — Pointers & Dynamic Memory

Announcements

- Exam 1 is on Monday Feb 6, at 6pm. Check Submitty for room assignments. They might be up already, if not they should be up by the end of today (Friday). See Lecture 5's notes for more exam-related announcements.
- The next homework will be checked for memory errors on the server. Run Dr. Memory or Valgrind on your code to detect memory errors. See http://www.cs.rpi.edu/academics/courses/spring16/csci1200/memory_debugging.php for more information on how to run Dr. Memory or valgrind.

Review from Lecture 5

- Pointer variables, arrays, pointer arithmetic and dereferencing, character arrays, and calling conventions.

Today's Lecture — Pointers and Dynamic Memory

- Arrays and pointers
- Different types of memory
- Dynamic allocation of arrays
- Memory Debuggers

6.1 Three Types of Memory

- Automatic memory: memory allocation inside a function when you create a variable. This allocates space for local variables in functions (on the *stack*) and deallocates it when variables go out of scope. For example:

```
int x;
double y;
```

- Static memory: variables allocated statically (with the keyword `static`). They are are not eliminated when they go out of scope. They retain their values, but are only accessible within the scope where they are defined.

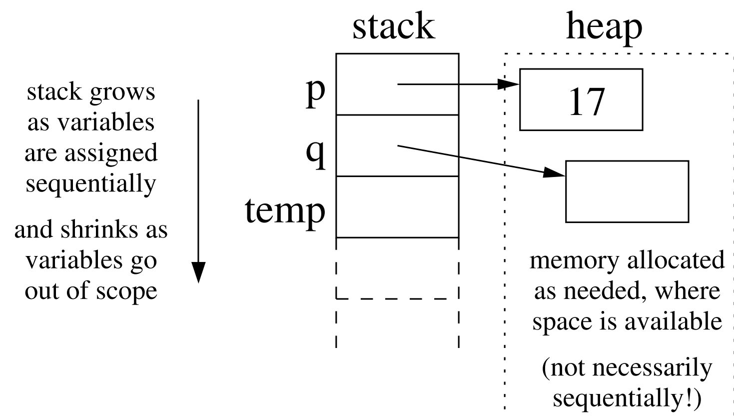
```
static int counter;
```

- Dynamic memory: explicitly allocated (on the *heap*) as needed. This is our focus for today.

6.2 Dynamic Memory

- Dynamic memory is:
  - created using the `new` operator,
  - accessed through pointers, and
  - removed through the `delete` operator.
- Here's a simple example involving dynamic allocation of integers:

```
int * p = new int;
*p = 17;
cout << *p << endl;
int * q;
q = new int;
*q = *p;
*p = 27;
cout << *p << " " << *q << endl;
int * temp = q;
q = p;
p = temp;
cout << *p << " " << *q << endl;
delete p;
delete q;
```



- The expression `new int` asks the system for a new chunk of memory that is large enough to hold an integer and returns the address of that memory. Therefore, the statement `int * p = new int;` allocates memory from the heap and stores its address in the pointer variable `p`.
- The statement `delete p;` takes the integer memory pointed by `p` and returns it to the system for re-use.
- This memory is allocated from and returned to a special area of memory called the *heap*. By contrast, local variables and function parameters are placed on the *stack* as discussed last lecture.
- In between the `new` and `delete` statements, the memory is treated just like memory for an ordinary variable, except the only way to access it is through pointers. Hence, the manipulation of pointer variables and values is similar to the examples covered in Lecture 5 except that there is no explicitly named variable for that memory other than the pointer variable.
- Dynamic allocation of primitives like ints and doubles is not very interesting or significant. What's more important is dynamic allocation of arrays and objects.

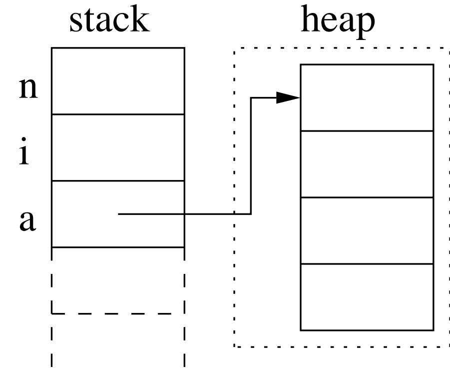
6.3 Exercise

- What's the output of the following code? Be sure to draw a picture to help you figure it out.

```
double * p = new double;
*p = 35.1;
double * q = p;
cout << *p << " " << *q << endl;
p = new double;
*p = 27.1;
cout << *p << " " << *q << endl;
*q = 12.5;
cout << *p << " " << *q << endl;
delete p;
delete q;
```

6.4 Dynamic Allocation of Arrays

- How do we allocate an array on the stack? What is the code? What memory diagram is produced by the code?
- Declaring the size of an array at compile time doesn't offer much flexibility. Instead we can *dynamically* allocate an array based on data. This gets us part-way toward the behavior of the standard library vector class. Here's an example:




```
int main() {
  std::cout << "Enter the size of the array: ";
  int n,i;
  std::cin >> n;
  double *a = new double[n];
  for (i=0; i<n; ++i)   { a[i] = sqrt(i); }
  for (i=0; i<n; ++i) {
    if ( double(int(a[i])) == a[i] )
      std::cout << i << " is a perfect square " << std::endl;
  }
  delete [] a;
  return 0;
}
```

- The expression `new double[n]` asks the system to *dynamically* allocate enough consecutive memory to hold n double's (usually $8n$ bytes).
  - What's crucially important is that `n` is a variable. Therefore, its value and, as a result, the size of the array are not known until the program is executed and the the memory must be allocated dynamically.
  - The address of the start of the allocated memory is assigned to the pointer variable `a`.
- After this, `a` is treated as though it is an array. For example: `a[i] = sqrt( i );`
  In fact, the expression `a[i]` is exactly equivalent to the pointer arithmetic and dereferencing expression `*(a+i)` which we have seen several times before.
- After we are done using the array, the line: `delete [] a;` releases the memory allocated for the entire array *and* calls the destructor (we'll learn about these soon!) for each slot of the array. Deleting a dynamically allocated array without the `[]` is an error (but it may not cause a crash or other noticeable problem, depending on the type stored in the array and the specific compiler implementation).
  - Since the program is ending, releasing the memory is not a major concern. However, to demonstrate that you understand memory allocation & deallocation, you should *always* delete dynamically allocated memory in this course, even if the program is terminating.
  - In more substantial programs it is ABSOLUTELY CRUCIAL. If we forget to release memory repeatedly the program can be said to have a *memory leak*. Long-running programs with memory leaks will eventually run out of memory and crash.

6.5 Exercises

1. Write code to dynamically allocate an array of `n` integers, point to this array using the integer pointer variable `a`, and then read n values into the array from the stream `cin`.

2. Now, suppose we wanted to write code to double the size of array `a` without losing the values. This requires some work: First allocate an array of size `2*n`, pointed to by integer pointer variable *temp* (which will become `a`). Then copy the `n` values of `a` into the first `n` locations of array `temp`. Finally delete array `a` and assign `temp` to `a`.

   Why don't you need to delete `temp`?

   Note: The code for part 2 of the exercise is very similar to what happens inside the `resize` member function of vectors!

6.6 Dynamic Allocation of Two-Dimensional Arrays

- To store a grid of data, we will need to allocate a top level array of pointers to arrays of the data. For example:

```
double** a = new double*[rows];
for (int i = 0; i < rows; i++) {
  a[i] = new double[cols];
  for (int j = 0; j < cols; j++) {
    a[i][j] = double(i+1) / double (j+1);
  }
}
```

  Draw a picture of the resulting data structure.
  Then, write code to correctly delete all of this memory.
  You need to call delete or delete [] as many times as you new or new [] respectively.

6.7 Dynamic Allocation: Arrays of Class Objects

- We can dynamically allocate arrays of class objects. The default constructor (the constructor that takes no arguments) must be defined in order to allocate an array of objects.

```
class Foo {
public:
  Foo();
  double value() const { return a*b; }
private:
  int a;
  double b;
};

Foo::Foo() {
  static int counter = 1;
  a = counter;
  b = 100.0;
  counter++;
}

int main() {
  int n;
  std::cin >> n;
  Foo *things = new Foo[n];
  std::cout << "size of int: " << sizeof(int) << std::endl;
  std::cout << "size of double: " << sizeof(double) << std::endl;
  std::cout << "size of foo object: " << sizeof(Foo) << std::endl;
  for (Foo* i = things; i < things+n; i++)
    std::cout << "Foo stored at: " << i << " has value " << i->value() << std::endl;
  delete [] things;
}
```

```
size of int: 4
size of double: 8
size of foo object: 16
Foo stored at: 0x104800890 has value 100
Foo stored at: 0x1048008a0 has value 200
Foo stored at: 0x1048008b0 has value 300
Foo stored at: 0x1048008c0 has value 400
...
```

- What does "->" do?
  It is a member access operator for objects created on the heap.
- We could also use (*i).value(). Why?

6.8 Memory Debugging

In addition to the step-by-step debuggers like `gdb`, `lldb`, or the debugger in your IDE, we recommend using a memory debugger like "Dr. Memory" (Windows, Linux, and MacOSX) or "Valgrind" (Linux and MacOSX). These tools can detect the following problems:

- Use of uninitialized memory
- Reading/writing memory after it has been free'd (*NOTE: delete calls free*)
- Reading/writing off the end of malloc'd blocks (*NOTE: new calls malloc*)
- Reading/writing inappropriate areas on the stack
- Memory leaks - where pointers to malloc'd blocks are lost forever
- Mismatched use of malloc/new/new [] vs free/delete/delete []
- Overlapping src and dst pointers in memcpy() and related functions

6.9 Sample Buggy Program

Can you see the errors in this program?

```
 1 #include <iostream>
 2
 3 int main() {
 4
 5   int *p = new int;
 6   if (*p != 10) std::cout << "hi" << std::endl;
 7
 8   int *a = new int[3];
 9   a[3] = 12;
10   delete a;
11
12 }
```

6.10 Using Dr. Memory `http://www.drmemory.org`

Here's how Dr. Memory reports the errors in the above program:

```
~~Dr.M~~ Dr. Memory version 1.8.0
~~Dr.M~~
~~Dr.M~~ Error #1: UNINITIALIZED READ: reading 4 byte(s)
~~Dr.M~~ # 0 main               [memory_debugger_test.cpp:6]
hi
~~Dr.M~~
~~Dr.M~~ Error #2: UNADDRESSABLE ACCESS beyond heap bounds: writing 4 byte(s)
~~Dr.M~~ # 0 main               [memory_debugger_test.cpp:9]
~~Dr.M~~ Note: refers to 0 byte(s) beyond last valid byte in prior malloc
~~Dr.M~~
~~Dr.M~~ Error #3: INVALID HEAP ARGUMENT: allocated with operator new[], freed with operator delete
~~Dr.M~~ # 0 replace_operator_delete              [/drmemory_package/common/alloc_replace.c:2684]
~~Dr.M~~ # 1 main                                 [memory_debugger_test.cpp:10]
~~Dr.M~~ Note: memory was allocated here:
~~Dr.M~~ Note: # 0 replace_operator_new_array              [/drmemory_package/common/alloc_replace.c:2638]
~~Dr.M~~ Note: # 1 main                                    [memory_debugger_test.cpp:8]
~~Dr.M~~
~~Dr.M~~ Error #4: LEAK 4 bytes
~~Dr.M~~ # 0 replace_operator_new               [/drmemory_package/common/alloc_replace.c:2609]
~~Dr.M~~ # 1 main                               [memory_debugger_test.cpp:5]
~~Dr.M~~
~~Dr.M~~ ERRORS FOUND:
~~Dr.M~~       1 unique,     1 total unaddressable access(es)
~~Dr.M~~       1 unique,     1 total uninitialized access(es)
```

```
~~Dr.M~~       1 unique,     1 total invalid heap argument(s)
~~Dr.M~~       0 unique,     0 total warning(s)
~~Dr.M~~       1 unique,     1 total,      4 byte(s) of leak(s)
~~Dr.M~~       0 unique,     0 total,      0 byte(s) of possible leak(s)
~~Dr.M~~ Details: /DrMemory-MacOS-1.8.0-8/drmemory/logs/DrMemory-a.out.7726.000/results.txt
```

And the fixed version:

```
~~Dr.M~~ Dr. Memory version 1.8.0
hi
~~Dr.M~~
~~Dr.M~~ NO ERRORS FOUND:
~~Dr.M~~       0 unique,     0 total unaddressable access(es)
~~Dr.M~~       0 unique,     0 total uninitialized access(es)
~~Dr.M~~       0 unique,     0 total invalid heap argument(s)
~~Dr.M~~       0 unique,     0 total warning(s)
~~Dr.M~~       0 unique,     0 total,      0 byte(s) of leak(s)
~~Dr.M~~       0 unique,     0 total,      0 byte(s) of possible leak(s)
~~Dr.M~~ Details: /DrMemory-MacOS-1.8.0-8/drmemory/logs/DrMemory-a.out.7762.000/results.txt
```

Note: Dr. Memory on Windows with the Visual Studio compiler may not report a mismatched free() / delete / delete [] error (e.g., line 10 of the sample code above). This may happen if optimizations are enabled and the objects stored in the array are simple and do not have their own dynamically-allocated memory that lead to their own indirect memory leaks.

6.11 Using Valgrind `http://valgrind.org/`

And this is how Valgrind reports the same errors:

```
==31226== Memcheck, a memory error detector
==31226== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==31226== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==31226== Command: ./a.out
==31226==
==31226== Conditional jump or move depends on uninitialised value(s)
==31226==    at 0x40096F: main (memory_debugger_test.cpp:6)
==31226==
hi
==31226== Invalid write of size 4
==31226==    at 0x4009A3: main (memory_debugger_test.cpp:9)
==31226==  Address 0x4c3f09c is 0 bytes after a block of size 12 alloc'd
==31226==    at 0x4A0700A: operator new[](unsigned long) (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==31226==    by 0x400996: main (memory_debugger_test.cpp:8)
==31226==
==31226== Mismatched free() / delete / delete []
==31226==    at 0x4A07991: operator delete(void*) (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==31226==    by 0x4009B4: main (memory_debugger_test.cpp:10)
==31226==  Address 0x4c3f090 is 0 bytes inside a block of size 12 alloc'd
==31226==    at 0x4A0700A: operator new[](unsigned long) (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==31226==    by 0x400996: main (memory_debugger_test.cpp:8)
==31226==
==31226==
==31226== HEAP SUMMARY:
==31226==     in use at exit: 4 bytes in 1 blocks
==31226==   total heap usage: 2 allocs, 1 frees, 16 bytes allocated
==31226==
==31226== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==31226==    at 0x4A06965: operator new(unsigned long) (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==31226==    by 0x400961: main (memory_debugger_test.cpp:5)
==31226==
==31226== LEAK SUMMARY:
==31226==    definitely lost: 4 bytes in 1 blocks
```

```
==31226==    indirectly lost: 0 bytes in 0 blocks
==31226==      possibly lost: 0 bytes in 0 blocks
==31226==    still reachable: 0 bytes in 0 blocks
==31226==         suppressed: 0 bytes in 0 blocks
==31226==
==31226== For counts of detected and suppressed errors, rerun with: -v
==31226== Use --track-origins=yes to see where uninitialised values come from
==31226== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 2 from 2)
```

And here's what it looks like after fixing those bugs:

```
==31252== Memcheck, a memory error detector
==31252== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==31252== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==31252== Command: ./a.out
==31252==
hi
==31252==
==31252== HEAP SUMMARY:
==31252==     in use at exit: 0 bytes in 0 blocks
==31252==   total heap usage: 2 allocs, 2 frees, 16 bytes allocated
==31252==
==31252== All heap blocks were freed -- no leaks are possible
==31252==
==31252== For counts of detected and suppressed errors, rerun with: -v
==31252== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

6.12 How to use a memory debugger

- Detailed instructions on installation & use of these tools are available here:
  http://www.cs.rpi.edu/academics/courses/spring17/ds/memory_debugging.php
- **Memory errors** (uninitialized memory, out-of-bounds read/write, use after free) may cause seg faults, crashes, or strange output.
- **Memory leaks** on the other hand will never cause incorrect output, but your program will be inefficient and hog system resources. A program with a memory leak may waste so much memory it causes all programs on the system to slow down significantly or it may crash the program or the whole operating system if the system runs out of memory (this takes a while on modern computers with lots of RAM & harddrive space).
- For HW3, the homework submission server will be configured to run your code with Dr. Memory to search for memory problems and present the output with the submission results. For full credit your program must be memory error and memory leak free!
- A program that seems to run perfectly fine on one computer may still have significant memory errors. Running a memory debugger will help find issues that might break your homework on another computer or when submitted to the homework server.
- **Important Note:** When these tool find a memory leak, they point to the line of code where this memory was *allocated*. These tools does not understand the program logic and thus obviously cannot tell us where it *should* have been deleted.
- A final note: STL and other 3rd party libraries are highly optimized and sometimes do sneaky but correct and bug-free tricks for efficiency that confuse the memory debugger. For example, because the STL string class uses its own allocator, there may be a warning about memory that is "still reachable" even though you've deleted all your dynamically allocated memory. The memory debuggers have automatic suppressions for some of these known "false positives", so you will see this listed as a "suppressed leak". So don't worry if you see those messages.

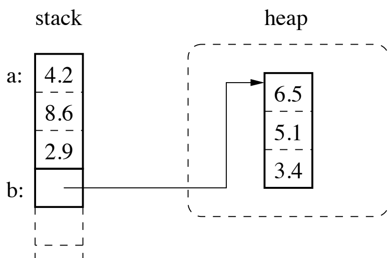
6.13 Diagramming Memory Exercises

- Draw a diagram of the *heap* and *stack* memory for each segment of code below. Use a "?" to indicate that the value of the memory is uninitialized. Indicate whether there are any errors or memory leaks during execution of this code.

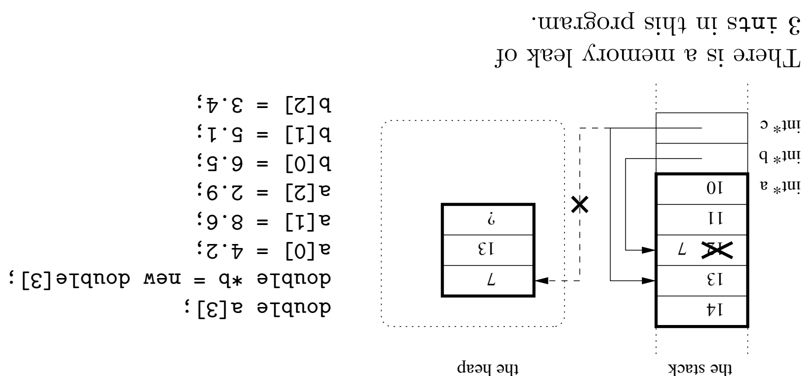
```
class Foo {
public:
  double x;
  int* y;
};
Foo a;
a.x = 3.14159;
Foo *b = new Foo;
(*b).y = new int[2];
Foo *c = b;
a.y = b->y;
c->y[1] = 7;
b = NULL;
```

```
int a[5] = { 10, 11, 12, 13, 14 };
int *b = a + 2;
*b = 7;
int *c = new int[3];
c[0] = b[0];
c[1] = b[1];
c = &(a[3]);
```

- Write code to produce this diagram:



6.14 Solutions to Diagramming Memory Exercises



CSCI-1200 Data Structures — Spring 2017
Lecture 7 — Order Notation & Basic Recursion

Review from Lectures 5 & 6

- Arrays and pointers, Pointer arithmetic and dereferencing
- Different types of memory ("automatic", static, dynamic)
- Dynamic allocation of arrays
- Drawing pictures to explain stack vs. heap memory allocation
- Memory debugging

Today's Lecture

- Algorithm Analysis
- Formal Definition of Order Notation
- Simple recursion
- Visualization of recursion
- Iteration vs. Recursion
- "Rules" for writing recursive functions.
- Lots of examples!

7.1 Algorithm Analysis

Why should we bother?

- We want to do better than just implementing and testing every idea we have.
- We want to know why one algorithm is better than another.
- We want to know the best we can do. (This is often quite hard.)

How do we do it? There are several options, including:

- Don't do any analysis; just use the first algorithm you can think of that works.
- Implement and time algorithms to choose the best.
- Analyze algorithms by counting operations while assigning different weights to different types of operations based on how long each takes.
- Analyze algorithms by assuming each operation requires the same amount of time. Count the total number of operations, and then multiply this count by the average cost of an operation.

7.2 Exercise: Counting Example

- Suppose `arr` is an array of `n` doubles. Here is a simple fragment of code to sum of the values in the array:

```
double sum = 0;
for (int i=0; i<n; ++i)
  sum += arr[i];
```

- What is the total number of operations performed in executing this fragment? Come up with a function describing the number of operations *in terms of n.*

7.3 Exercise: Which Algorithm is Best?

A venture capitalist is trying to decide which of 3 startup companies to invest in and has asked for your help. Here's the timing data for their prototype software on some different size test cases:

n	foo-a	foo-b	foo-c
10	10 u-sec	5 u-sec	1 u-sec
20	13 u-sec	10 u-sec	8 u-sec
30	15 u-sec	15 u-sec	27 u-sec
100	20 u-sec	50 u-sec	1000 u-sec
1000	?	?	?

Which company has the "best" algorithm?

7.4 Order Notation Definition

In this course we will focus on the intuition of order notation. This topic will be covered again, in more depth, in later computer science courses.

- Definition: Algorithm A is order $f(n)$ — denoted $O(f(n))$ — if constants k and n_0 exist such that A requires no more than $k * f(n)$ time units (operations) to solve a problem of size $n \geq n_0$.
- For example, algorithms requiring $3n + 2$, $5n - 3$, and $14 + 17n$ operations are all $O(n)$.

  This is because we can select values for k and n_0 such that the definition above holds. (What values?)

  Likewise, algorithms requiring $n^2/10 + 15n - 3$ and $10000 + 35n^2$ are all $O(n^2)$.
- Intuitively, we determine the order by finding the *asymptotically dominant term (function of n)* and throwing out the leading constant. This term could involve logarithmic or exponential functions of n. Implications for analysis:
  - We don't need to quibble about small differences in the numbers of operations.
  - We also do not need to worry about the different costs of different types of operations.
  - We don't produce an actual time. We just obtain a rough count of the number of operations. This count is used for comparison purposes.
- In practice, this makes analysis relatively simple, quick and (sometimes unfortunately) rough.

7.5 Common Orders of Magnitude

- $O(1)$, *a.k.a. CONSTANT*: The number of operations is independent of the size of the problem. e.g., compute quadratic root.
- $O(\log n)$, *a.k.a. LOGARITHMIC.* e.g., dictionary lookup, binary search.
- $O(n)$, *a.k.a. LINEAR.* e.g., sum up a list.
- $O(n \log n)$, e.g., sorting.
- $O(n^2)$, $O(n^3)$, $O(n^k)$, *a.k.a. POLYNOMIAL.* e.g., find closest pair of points.
- $O(2^n)$, $O(k^n)$, *a.k.a. EXPONENTIAL.* e.g., Fibonacci, playing chess.

7.6 Exercise: A Slightly Harder Example

- Here's an algorithm to determine if the value stored in variable `x` is also in an array called `foo`. Can you analyze it? What did you do about the `if` statement? What did you assume about where the value stored in `x` occurs in the array (if at all)?

```
int loc=0;
bool found = false;
while (!found && loc < n) {
  if (x == foo[loc]) found = true;
  else loc++;
}
if (found) cout << "It is there!\n";
```

7.7 Best-Case, Average-Case and Worst-Case Analysis

- For a given fixed size array, we might want to know:
  - The fewest number of operations (best case) that might occur.
  - The average number of operations (average case) that will occur.
  - The maximum number of operations (worst case) that can occur.
- The last is the most common. The first is rarely used.
- On the previous algorithm, the best case is $O(1)$, but the average case and worst case are both $O(n)$.

7.8 Approaching An Analysis Problem

- Decide the important variable (or variables) that determine the "size" of the problem. For arrays and other "container classes" this will generally be the number of values stored.
- Decide what to count. The order notation helps us here.
  - If each loop iteration does a fixed (or bounded) amount of work, then we only need to count the number of loop iterations.
  - We might also count specific operations. For example, in the previous exercise, we could count the number of comparisons.
- Do the count and use order notation to describe the result.

7.9 Exercise: Order Notation

For each version below, give an order notation estimate of the number of operations as a function of `n`:

1.
```
int count=0;
for (int i=0; i<n; ++i)
  for (int j=0; j<n; ++j)
    ++count;
```

2.
```
int count=0;
for (int i=0; i<n; ++i)
  ++count;
for (int j=0; j<n; ++j)
  ++count;
```

3.
```
int count=0;
for (int i=0; i<n; ++i)
  for (int j=i; j<n; ++j)
    ++count;
```

7.10 Recursive Definitions of Factorials and Integer Exponentiation

- Factorial is defined for non-negative integers as

$$n! = \begin{cases} n \cdot (n-1)! & n > 0 \\ 1 & n == 0 \end{cases}$$

- Computing integer powers is defined as:

$$n^p = \begin{cases} n \cdot n^{p-1} & p > 0 \\ 1 & p == 0 \end{cases}$$

- These are both examples of *recursive definitions*.

7.11 Recursive C++ Functions

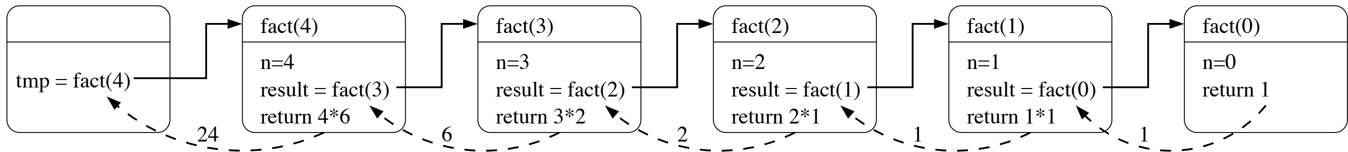
C++, like other modern programming languages, allows functions to call themselves. This gives a direct method of implementing recursive functions. Here are the recursive implementations of factorial and integer power:

```
int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    int result = fact(n-1);
    return n * result;
  }
}
```

```
int intpow(int n, int p) {
  if (p == 0) {
    return 1;
  } else {
    return n * intpow( n, p-1 );
  }
}
```

7.12 The Mechanism of Recursive Function Calls

- For each recursive call (or any function call), a program creates an *activation record* to keep track of:
  - **Completely separate instances** of the parameters and local variables for the newly-called function.
  - The location in the calling function code to return to when the newly-called function is complete. (Who asked for this function to be called? Who wants the answer?)
  - Which activation record to return to when the function is done. For recursive functions this can be confusing since there are multiple activation records waiting for an answer from the same function.
- This is illustrated in the following diagram of the call `fact(4)`. Each box is an activation record, the solid lines indicate the function calls, and the dashed lines indicate the returns. Inside of each box we list the parameters and local variables and make notes about the computation.



- This chain of activation records is stored in a special part of program memory called *the stack*.

7.13 Iteration vs. Recursion

- Each of the above functions could also have been written using a `for` or `while` loop, i.e. *iteratively*. For example, here is an iterative version of factorial:

```
int ifact(int n) {
  int result = 1;
  for (int i=1; i<=n; ++i)
    result = result * i;
  return result;
}
```

- Often writing recursive functions is more natural than writing iterative functions, especially for a first draft of a problem implementation.
- You should learn how to recognize whether an implementation is recursive or iterative, and practice rewriting one version as the other. Note: We'll see that not all recursive functions can be *easily* rewritten in iterative form!
- Note: The order notation for the number of operations for the recursive and iterative versions of an algorithm is usually the same. However in C, C++, Java, and some other languages, *iterative functions are generally faster than their corresponding recursive functions*. This is due to the overhead of the function call mechanism. Compiler optimizations will sometimes (but not always!) reduce the performance hit by automatically eliminating the recursive function calls. This is called *tail call optimization*.

7.14 Exercises

1. Draw a picture to illustrate the activation records for the function call

   ```
   cout << intpow(4, 4) << endl;
   ```

2. Write an iterative version of `intpow`.

7.15 Rules for Writing Recursive Functions

Here is an outline of five steps that are useful in writing and debugging recursive functions. Note: You don't have to do them in exactly this order...

1. Handle the base case(s).
2. Define the problem solution in terms of smaller instances of the problem. Use *wishful thinking*, i.e., if someone else solves the problem of `fact(4)` I can extend that solution to solve `fact(5)`. This defines the necessary recursive calls. It is also the hardest part!
3. Figure out what work needs to be done before making the recursive call(s).
4. Figure out what work needs to be done after the recursive call(s) complete(s) to finish the computation. (What are you going to do with the result of the recursive call?)
5. Assume the recursive calls work correctly, but make sure they are progressing toward the base case(s)!

7.16 Recursion Example: Printing the Contents of a Vector

- Here is a function to print the contents of a vector. Actually, it's two functions: a *driver function*, and a true recursive function. It is common to have a driver function that just initializes the first recursive function call.

```
void print_vec(std::vector<int>& v, unsigned int i) {
  if (i < v.size()) {
    cout << i << ": " << v[i] << endl;
    print_vec(v, i+1);
  }
}
void print_vec(std::vector<int>& v) {
  print_vec(v, 0);
}
```

- **Exercise:** What will this print when called in the following code?

```
int main() {
  std::vector<int> a;
  a.push_back(3);  a.push_back(5);  a.push_back(11); a.push_back(17);
  print_vec(a);
}
```

- **Exercise:** How can you change the second `print_vec` function as little as possible so that this code prints the contents of the vector in reverse order?

7.17 Binary Search

- Suppose you have a `std::vector<T> v` (for a placeholder type T), sorted so that:

  `v[0] <= v[1] <= v[2] <= ...`

- Now suppose that you want to find if a particular value `x` is in the vector somewhere. How can you do this without looking at every value in the vector?
- The solution is a recursive algorithm called *binary search*, based on the idea of checking the middle item of the search interval within the vector and then looking either in the lower half or the upper half of the vector, depending on the result of the comparison.

```
template <class T>
bool binsearch(const std::vector<T> &v, int low, int high, const T &x) {
 if (high == low) return x == v[low];
  int mid = (low+high) / 2;
  if (x <= v[mid])
    return binsearch(v, low, mid, x);
  else
    return binsearch(v, mid+1, high, x);
}
template <class T>
bool binsearch(const std::vector<T> &v, const T &x) {
  return binsearch(v, 0, v.size()-1, x);
}
```

7.18 Exercises

1. Write a non-recursive version of binary search.
2. If we replaced the if-else structure inside the recursive binsearch function (above) with

   ```
   if ( x < v[mid] )
     return binsearch( v, low, mid-1, x );
   else
     return binsearch( v, mid, high, x );
   ```

   would the function still work correctly?

CSCI-1200 Data Structures — Spring 2017
Lecture 8 — Templated Classes & Vector Implementation

Review from Lectures 7

- Algorithm Analysis, Formal Definition of Order Notation
- Simple recursion, Visualization of recursion, Iteration vs. Recursion, "Rules" for writing recursive functions.
- Lots of examples!

8.1 Today's Lecture

- Designing our own container classes:
  - Mimic the interface of standard library (STL) containers
  - Study the design of memory management.
  - Move toward eventually designing our own, more sophisticated classes.
- Vector implementation
- Templated classes (*including* compilation and instantiation of templated classes)
- Copy constructors, assignment operators, and destructors

Optional Reading: Ford&Topp, Sections 5.3-5.5; Koening & Moo Chapter 11

8.2 Vector Public Interface

- In creating our own version of the STL vector class, we will start by considering the public interface:

```
public:
  // MEMBER FUNCTIONS AND OTHER OPERATORS
  T& operator[] (size_type i);
  const T& operator[] (size_type i) const;
  void push_back(const T& t);
  void resize(size_type n, const T& fill_in_value = T());
  void clear();
  bool empty() const;
  size_type size() const;
```

- To implement our own generic (a.k.a. templated) vector class, we will implement all of these operations, manipulate the underlying representation, and discuss memory management.

8.3 Templated Class Declarations and Member Function Definitions

- In terms of the layout of the code in `vec.h` (pages 5 & 6 of the handout), the biggest difference is that this is a *templated class*. The keyword `template` and the template type name must appear before the class declaration:

```
template <class T> class Vec
```

- Within the class declaration, `T` is used as a type and all member functions are said to be "templated over type `T`". In the actual text of the code files, templated member functions are often defined (written) *inside the class declaration.*
- The templated functions defined outside the template class declaration must be preceded by the phrase: `template <class T>` and then when `Vec` is referred to it must be as `Vec<T>` . For example, for member function `create` (two versions), we write:

```
template <class T> void Vec<T>::create(...
```

8.4 Syntax and Compilation

- Templated classes and templated member functions are not created/compiled/instantiated until they are needed. Compilation of the class declaration is triggered by a line of the form: Vec<int> v1; with int replacing T. This also compiles the default constructor for Vec<int> because it is used here. Other member functions are not compiled unless they are used.
- When a different type is used with Vec, for example in the declaration: Vec<double> z; the template class declaration is compiled again, this time with double replacing T instead of int. Again, however, only the member functions used are compiled.
- This is very different from ordinary classes, which are usually compiled separately and all functions are compiled regardless of whether or not they are needed.
- The templated class declaration and the code for all used member functions must be provided where they are used. As a result, member functions definitions are often included within the class declaration or defined outside of the class declaration but still in the .h file. If member function definitions are placed in a separate .cpp file, this file must be #include-d, just like the .h file, because the compiler needs to see it in order to generate code. (Normally we don't #include .cpp files!) See also diagram on page 7 of this handout.

  Note: Including function definitions in the .h for ordinary non-templated classes may lead to compilation errors about functions being "multiply defined". Some of you have already seen these errors.

8.5 Member Variables

Now, looking inside the Vec<T> class at the member variables:

- m_data is a pointer to the start of the array (after it has been allocated). Recall the close relationship between pointers and arrays.
- m_size indicates the number of locations currently in use in the vector. This is exactly what the size() member function should return,
- m_alloc is the total number of slots in the dynamically allocated block of memory.

Drawing pictures, which we will do in class, will help clarify this, especially the distinction between m_size and m_alloc.

8.6 Typedefs

- Several types are created through typedef statements in the first public area of Vec. Once created the names are used as ordinary type names. For example Vec<int>::size_type is the return type of the size() function, defined here as an unsigned int.

8.7 operator[]

- Access to the individual locations of a Vec is provided through operator[]. Syntactically, use of this operator is translated by the compiler into a call to a function called operator[]. For example, if v is a Vec<int>, then:

  ```
  v[i] = 5;
  ```

  translates into:

  ```
  v.operator[](i) = 5;
  ```

- In most classes there are two versions of operator[]:
  - A non-const version returns a reference to m_data[i]. This is applied to non-const Vec objects.
  - A const version is the one called for const Vec objects. This also returns m_data[i], but as a const reference, so it can not be modified.

8.8 Default Versions of Assignment Operator and Copy Constructor Are Dangerous!

- Before we write the copy constructor and the assignment operator, we consider what would happen if we didn't write them.
- C++ compilers provide default versions of these if they are not provided. These defaults just copy the values of the member variables, one-by-one. For example, the default copy constructor would look like this:

```
template <class T>
Vec<T> :: Vec(const Vec<T>& v)
 : m_data(v.m_data), m_size(v.m_size), m_alloc(v.m_alloc)
{}
```

  In other words, it would construct each member variable from the corresponding member variable of `v`. This is dangerous and incorrect behavior for the `Vec` class. We don't want to just copy the `m_data` pointer. We really want to create a copy of the entire array! Let's look at this more closely...

8.9 Exercise

Suppose we used the default version of the assignment operator and copy constructor in our `Vec<T>` class. What would be the output of the following program? Assume all of the operations **except** the copy constructor behave as they would with a `std::vector<double>`.

```
Vec<double> v(4, 0.0);
v[0] = 13.1;  v[2] = 3.14;
Vec<double> u(v);
u[2] = 6.5;
u[3] = -4.8;
for (unsigned int i=0; i<4; ++i)
  cout << u[i] << " " << v[i] << endl;
```

Explain what happens by drawing a picture of the memory of both `u` and `v`.

8.10 Classes With Dynamically Allocated Memory

- For `Vec` (and other classes with dynamically-allocated memory) to work correctly, each object must do its own dynamic memory allocation and deallocation. We must be careful to keep the memory of each object instance separate from all others.
- All dynamically-allocated memory for an object should be released when the object is finished with it or when the object itself goes out of scope (through what's called a *destructor*).
- To prevent the creation and use of default versions of these operations, we must write our own:
  - *Copy constructor*
  - *Assignment operator*
  - *Destructor*

8.11 The "this" pointer

- All class objects have a special pointer defined called `this` which simply points to the current class object, and it may not be changed.
- The expression `*this` is a reference to the class object.
- The `this` pointer is used in several ways:
  - Make it clear when member variables of the current object are being used.
  - Check to see when an assignment is self-referencing.
  - Return a reference to the current object.

8.12 Copy Constructor

- This constructor must dynamically allocate any memory needed for the object being constructed, copy the contents of the memory of the passed object to this new memory, and set the values of the various member variables appropriately.

- **Exercise:** In our `Vec` class, the actual copying is done in a private member function called `copy`. Write the private member function `copy`.

8.13 Assignment Operator

- Assignment operators of the form: `v1 = v2;`

  are translated by the compiler as: `v1.operator=(v2);`

- Cascaded assignment operators of the form: `v1 = v2 = v3;`

  are translated by the compiler as: `v1.operator=(v2.operator=(v3));`

- Therefore, the value of the assignment operator (`v2 = v3`) must be suitable for input to a second assignment operator. This in turn means the result of an assignment operator ought to be a reference to an object.

- The implementation of an assignment operator usually takes on the same form for every class:
  - Do no real work if there is a self-assignment.
  - Otherwise, destroy the contents of the current object then copy the passed object, just as done by the copy constructor. In fact, it often makes sense to write a private helper function used by both the copy constructor and the assignment operator.
  - Return a reference to the (copied) current object, using the `this` pointer.

8.14 Destructor (the "constructor with a tilde/twiddle")

- The destructor is called implicitly when an automatically-allocated object goes *out of scope* or a dynamically-allocated object is *deleted*. It can never be called explicitly!

- The destructor is responsible for deleting the dynamic memory "owned" by the class.

- The syntax of the function definition is a bit weird. The `~` has been used as a logic negation in other contexts.

8.15 Increasing the Size of the Vec

- `push_back(T const& x)` adds to the end of the array, increasing `m_size` by one `T` location. But what if the allocated array is full (`m_size == m_alloc`)?
  1. Allocate a new, larger array. The best strategy is generally to double the size of the current array. Why?
  2. If the array size was originally 0, doubling does nothing. We must be sure that the resulting size is at least 1.
  3. Then we need to copy the contents of the current array.
  4. Finally, we must delete current array, make the `m_data` pointer point to the start of the new array, and adjust the `m_size` and `m_alloc` variables appropriately.

- Only when we are sure there is enough room in the array should we actually add the new object to the back of the array.

8.16 Exercises

- Finish the definition of `Vec::push_back`.

- Write the `Vec::resize` function.

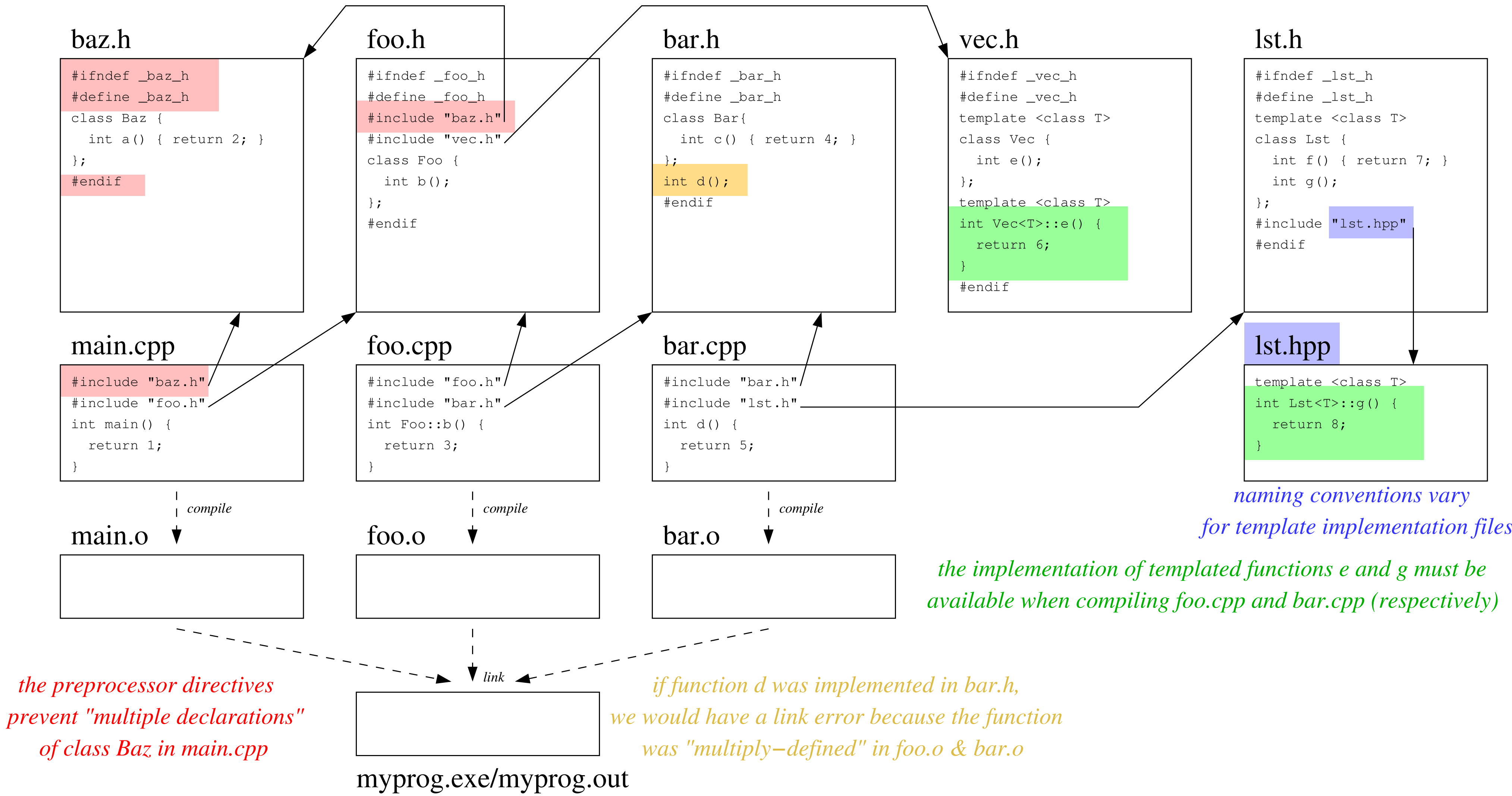
8.17 Vec Declaration & Implementation (vec.h)

```
#ifndef Vec_h_
#define Vec_h_
// Simple implementation of the vector class, revised from Koenig and Moo.  This
// class is implemented using a dynamically allocated array (of templated type T).
// We ensure that that m_size is always <= m_alloc and when a push_back or resize
// call would violate this condition, the data is copied to a larger array.

template <class T> class Vec {

public:
  // TYPEDEFS
  typedef unsigned int size_type;

  // CONSTRUCTORS, ASSIGNMNENT OPERATOR, & DESTRUCTOR
  Vec() { this->create(); }
  Vec(size_type n, const T& t = T()) { this->create(n, t); }
  Vec(const Vec& v) { copy(v); }
  Vec& operator=(const Vec& v);
  ~Vec() { delete [] m_data; }

  // MEMBER FUNCTIONS AND OTHER OPERATORS
  T& operator[] (size_type i) { return m_data[i]; }
  const T& operator[] (size_type i) const { return m_data[i]; }
  void push_back(const T& t);
  void resize(size_type n, const T& fill_in_value = T());
  void clear() { delete [] m_data;  create(); }
  bool empty() const { return m_size == 0; }
  size_type size() const { return m_size; }

private:
  // PRIVATE MEMBER FUNCTIONS
  void create();
  void create(size_type n, const T& val);
  void copy(const Vec<T>& v);

  // REPRESENTATION
  T* m_data;         // Pointer to first location in the allocated array
  size_type m_size;  // Number of elements stored in the vector
  size_type m_alloc; // Number of array locations allocated,  m_size <= m_alloc
};

// Create an empty vector (null pointers everywhere).
template <class T>  void Vec<T>::create() {
  m_data = NULL;
  m_size = m_alloc = 0;  // No memory allocated yet
}

// Create a vector with size n, each location having the given value
template <class T> void Vec<T>::create(size_type n, const T& val) {
  m_data = new T[n];
  m_size = m_alloc = n;
  for (size_type i = 0; i < m_size; i++) {
    m_data[i] = val;
  }
}

// Assign one vector to another, avoiding duplicate copying.
template <class T> Vec<T>& Vec<T>::operator=(const Vec<T>& v) {
  if (this != &v) {
    delete [] m_data;
    this -> copy(v);
  }
  return *this;
}
```

```
// Create the vector as a copy of the given vector.
template <class T> void Vec<T>::copy(const Vec<T>& v) {









}

// Add an element to the end, resize if necesssary.
template <class T> void Vec<T>::push_back(const T& val) {
  if (m_size == m_alloc) {
    // Allocate a larger array, and copy the old values











  }
  // Add the value at the last location and increment the bound
  m_data[m_size] = val;
  ++ m_size;
}

// If n is less than or equal to the current size, just change the size.  If n is
// greater than the current size, the new slots must be filled in with the given value.
// Re-allocation should occur only if necessary.  push_back should not be used.
template <class T> void Vec<T>::resize(size_type n, const T& fill_in_value) {

















}

#endif
```

8.18 File Organization & Compilation of Templated Classes

The diagram on the next page shows the typical and suggested file organization for non-templated vs. templated classes. Common mistakes and the resulting compilation errors are noted.



baz.h
#ifndef _baz_h
#define _baz_h
class Baz {
int a() { return 2; }
};
#endif
foo.h
#ifndef _foo_h
#define _foo_h
#include "baz.h"
#include "vec.h"
class Foo {
int b();
};
#endif
bar.h
#ifndef _bar_h
#define _bar_h
class Bar{
int c() { return 4; }
};
int d();
#endif
vec.h
#ifndef _vec_h
#define _vec_h
template <class T>
class Vec {
int e();
};
template <class T>
int Vec<T>::e() {
return 6;
}
#endif
lst.h
#ifndef _lst_h
#define _lst_h
template <class T>
class Lst {
int f() { return 7; }
int g();
};
#include "lst.hpp"
#endif
main.cpp
#include "baz.h"
#include "foo.h"
int main() {
return 1;
}
foo.cpp
#include "foo.h"
#include "bar.h"
int Foo::b() {
return 3;
}
bar.cpp
#include "bar.h"
#include "lst.h"
int d() {
return 5;
}
lst.hpp
template <class T>
int Lst<T>::g() {
return 8;
}
compile
compile
compile
main.o
foo.o
bar.o
naming conventions vary
for template implementation files
the implementation of templated functions e and g must be
available when compiling foo.cpp and bar.cpp (respectively)
link
the preprocessor directives
prevent "multiple declarations"
of class Baz in main.cpp
if function d was implemented in bar.h,
we would have a link error because the function
was "multiply−defined" in foo.o & bar.o
myprog.exe/myprog.out

CSCI-1200 Data Structures — Spring 2017
Lecture 9 — Iterators & STL Lists

Review from Lecture 8

- Designing our own container classes
- Dynamically allocated memory in classes
- Copy constructors, assignment operators, and destructors
- Templated classes, Implementation of the DS `Vec` class, mimicking the STL `vector` class

HW3 Tips

- You must write the assignment operator,
`Matrix::operator=(const Matrix& other_matrix)`
- When writing copy constructors and assignment operators, if there is dynamic memory involved, you must copy the **values**, not the **pointers**.
- Draw memory diagrams! Use small matrices (the SimpleTest() matrices are all small) so that you can draw out the details. Follow your code line by line.
- The homework assignment shows how the matrix data is organized in a `double**`. Which part(s) are on the stack and which are on the heap?
- If an assertion fails, your code will crash. This is by design. Fine the line number of the assertion, and see what the assert was testing. Read the lines above it too.
- Use Dr. Memory or Valgrind to catch leaks and memory errors. Not fixing these can lead to problems all over.
- Let's consider quarter() of a 1x1 and of a 0x0 together.

Today

- Another `vector` operation: `pop_back`
- *Erasing items* from vectors is inefficient!
- Iterators and iterator operations
- STL `lists` are a different sequential container class.
- Returning references to member variables from member functions
- `Vec` iterator implementation

Optional Reading: Ford & Topp Ch 6; Koenig & Moo, Sections 5.1-5.5

9.1 Review: Constructors, Assignment Operator, and Destructor

From an old test: Match up the line of code with the function that is called. Each letter is used exactly once.

	Code		Function
☐	`Foo f1;`	a)	assignment operator
☐	`Foo* f2;`	b)	destructor
☐	`f2 = new Foo(f1);`	c)	copy constructor
☐	`f1 = *f2;`	d)	default constructor
☐	`delete f2;`	e)	none of the above

9.2 Another STL `vector` operation: `pop_back`

- We have seen how `push_back` adds a value to the end of a vector, increasing the size of the vector by 1. There is a corresponding function called `pop_back`, which removes the last item in a vector, reducing the size by 1.
- There are also vector functions called `front` and `back` which denote (and thereby provide access to) the first and last item in the vector, allowing them to be changed. For example:

```
vector<int> a(5,1); // a has 5 values, all 1
a.pop_back();       // a now has 4 values
a.front() = 3;      // equivalent to the statement, a[0] = 3;
a.back() = -2;      // equivalent to the statement, a[a.size()-1] = -2;
```

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <assert.h>
using namespace std;

void erase_from_vector(unsigned int i, vector<string>& v) {
  /* EXERCISE: IMPLEMENT THIS FUNCTION */
}

// Enroll a student if there is room and the student is not already in course or on waiting list.
void enroll_student(const string& id, unsigned int max_students,
                    vector<string>& enrolled, vector<string>& waiting) {
  // Check to see if the student is already enrolled.
  unsigned int i;
  for (i=0; i < enrolled.size(); ++i) {
    if (enrolled[i] == id) {
      cout << "Student " << id << " is already enrolled." << endl;
      return;
    }
  }
  // If the course isn't full, add the student.
  if (enrolled.size() < max_students) {
    enrolled.push_back(id);
    cout << "Student " << id << " added.\n"
         << enrolled.size() << " students are now in the course." << endl;
    return;
  }
  // Check to see if the student is already on the waiting list.
  for (i=0; i < waiting.size(); ++i) {
    if (waiting[i] == id) {
      cout << "Student " << id << " is already on the waiting list." << endl;
      return;
    }
  }
  // If not, add the student to the waiting list.
  waiting.push_back(id);
  cout << "The course is full.  Student " << id << " has been added to the waiting list.\n"
       << waiting.size() << " students are on the waiting list." << endl;
}

// Remove a student from the course or from the waiting list.  If removing the student from the
// course opens up a slot, then the first person on the waiting list is placed in the course.
void remove_student(const string& id, unsigned int max_students,
                    vector<string>& enrolled, vector<string>& waiting) {
  // Check to see if the student is on the course list.
  bool found = false;
  unsigned int loc=0;
  while (!found && loc < enrolled.size()) {
    found = enrolled[loc] == id;
    if (!found) ++loc;
  }
  if (found) {
    // Remove the student and see if a student can be taken from the waiting list.
    erase_from_vector(loc, enrolled);
    cout << "Student " << id << " removed from the course." << endl;
    if (waiting.size() > 0) {
      enrolled.push_back(waiting[0]);
      cout << "Student " << waiting[0] << " added to the course from the waiting list." << endl;
      erase_from_vector(0, waiting);
      cout << waiting.size() << " students remain on the waiting list." << endl;
    } else {
      cout << enrolled.size() << " students are now in the course." << endl;
    }
  } else {
    // Check to see if the student is on the waiting list
    found = false;
    loc = 0;
    while (!found && loc < waiting.size()) {
      found = waiting[loc] == id;
      if (!found) ++loc;
    }
    if (found) {
      erase_from_vector(loc, waiting);
      cout << "Student " << id << " removed from the waiting list.\n"
           << waiting.size() << " students remain on the waiting list." << endl;
    } else {
      cout << "Student " << id << " is in neither the course nor the waiting list" << endl;
    }
  }
}

int main() {
  // Read in the maximum number of students in the course
  unsigned int max_students;
  cout << "\nEnrollment program for CSCI 1200\nEnter the maximum number of students allowed\n";
  cin >> max_students;

  // Initialize the vectors
  vector<string> enrolled;
  vector<string> waiting;

  // Invariant:
  // (1) enrolled contains the students already in the course,
  // (2) waiting contains students who will be admitted (in the order of request) if a spot opens up
  // (3) enrolled.size() <= max_students,
  // (4) if the course is not filled (enrolled.size() != max_students) then waiting is empty
  do {
    // check (part of) the invariant
    assert (enrolled.size() <= max_students);
    assert (enrolled.size() == max_students || waiting.size() == 0);
    cout << "\nOptions:\n"
         << "  To enroll a student type 0\n"
         << "  To remove a student type 1\n"
         << "  To end type 2\n"
         << "Type option ==> ";
    int option;
    if (!(cin >> option)) { // if we can't read the input integer, then just fail.
      cout << "Illegal input.  Good-bye.\n";
      return 1;
    } else if (option == 2) {
      break; // quit by breaking out of the loop.
    } else if (option != 0 && option != 1) {
      cout << "Invalid option.  Try again.\n";
    } else { // option is 0 or 1
      string id;
      cout << "Enter student id: ";
      if (!(cin >> id)) {
        cout << "Illegal input.  Good-bye.\n";
        return 1;
      } else if (option == 0) {
        enroll_student(id, max_students, enrolled, waiting);
      } else {
        remove_student(id, max_students, enrolled, waiting);
      }
    }
  }
  while (true);

  // some nice output
  sort(enrolled.begin(), enrolled.end());
  cout << "\nAt the end of the enrollment period, the following students are in the class:\n\n";
  for (unsigned int i=0; i<enrolled.size(); ++i) { cout << enrolled[i] << endl;  }
  if (!waiting.empty()) {
    cout << "\nStudents are on the waiting list in the following order:\n";
    for (unsigned int j=0; j<waiting.size(); ++j) { cout << waiting[j] << endl; }
  }
  return 0;
}
```

9.3 Motivating Example: Course Enrollment and Waiting List

- This program maintains the class list and the waiting list for a single course. The program is structured to handle interactive input. Error checking ensures that the input is valid.
- Vectors store the enrolled students and the waiting students. The main work is done in the two functions `enroll_student` and `remove_student`.
- The invariant on the loop in the main function determines how these functions must behave.

9.4 Exercises

1. Write `erase_from_vector`. This function removes the value at index location i from a vector of strings. The size of the vector should be reduced by one when the function is finished.

```
// Remove the value at index location i from a vector of strings.  The
// size of the vector should be reduced by one when the function is finished.
void erase_from_vector(unsigned int i,  vector<string>& v) {




}
```

2. Give an order notation estimate of the average cost of `erase_from_vector`, `pop_back`, and `push_back`.

9.5 What To Do About the Expense of Erasing From a Vector?

- When items are continually being inserted and removed, vectors are not a good choice for the container.
- Instead we need a different sequential container, called a *list*.
  - This has a "linked" structure that makes the cost of erasing independent of the size.
- We will move toward a list-based implementation of the program in two steps:
  - Rewriting our `classlist_vec.cpp` code in terms of *iterator* operations.
  - Replacing vectors with lists

9.6 Iterators

- Here's the definition (from Koenig & Moo). An iterator:
  - identifies a container and a specific element stored in the container,
  - lets us examine (and change, except for const iterators) the value stored at that element of the container,
  - provides operations for moving (the iterators) between elements in the container,
  - restricts the available operations in ways that correspond to what the container can handle efficiently.
- As we will see, iterators for different container classes have many operations in common. This often makes the switch between containers fairly straightforward from the programer's viewpoint.
- Iterators in many ways are generalizations of pointers: many operators / operations defined for pointers are defined for iterators. You should use this to guide your beginning understanding and use of iterators.

9.7 Iterator Declarations and Operations

- Iterator types are declared by the container class. For example,

  ```
  vector<string>::iterator p;
  vector<string>::const_iterator q;
  ```

  defines two (uninitialized) iterator variables.
- The *dereference operator* is used to access the value stored at an element of the container. The code:

  ```
  p = enrolled.begin();
  *p = "012312";
  ```

  changes the first entry in the `enrolled` vector.

- The dereference operator is combined with dot operator for accessing the member variables and member functions of elements stored in containers. Here's an example using the `Student` class and `students` vector from Lecture 4:

```
vector<Student>::iterator i = students.begin();
(*i).compute_averages(0.45);
```

  Notes:

  – This operation would be illegal if `i` had been defined as a `const_iterator` because `compute_averages` is a non-const member function.

  – The parentheses on the `*i` are **required** (because of operator precedence).

- There is a "syntactic sugar" for the combination of the dereference operator and the dot operator, which is exactly equivalent:

```
vector<StudentRec>::iterator i = students.begin();
i->compute_averages(0.45);
```

- Just like pointers, iterators can be incremented and decremented using the `++` and `--` operators to move to the next or previous element of any container.
- Iterators can be compared using the `==` and `!=` operators.
- Iterators can be assigned, just like any other variable.
- Vector iterators have several additional operations:
  – Integer values may be added to them or subtracted from them. This leads to statements like

```
enrolled.erase(enrolled.begin() + 5);
```

  – Vector iterators may be compared using operators like `<`, `<=`, etc.

  – For most containers (other than vectors), these "random access" iterator operations are not legal and therefore prevented by the compiler. The reasons will become clear as we look at their implementations.

9.8 Exercise: Revising the Class List Program to Use Iterators

- Now let's modify the class list program to use iterators. First rewrite the `erase_from_vector` to use iterators.

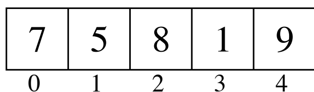
```
void erase_from_vector(vector<string>::iterator itr,  vector<string>& v) {







}
```

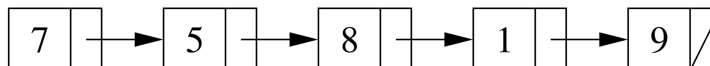
Note: the STL `vector` *class has a function that does just this... called* `erase`*!*

- Now, edit the rest of the file to remove all use of the vector subscripting operator.

9.9 A New Datatype: The `list` Standard Library Container Class

- Lists are our second standard-library container class. (Vectors were the first.) Both lists & vectors store sequential data that can shrink or grow.
- However, the use of memory is fundamentally different. Vectors are formed as a single contiguous array-like block of memory. Lists are formed as a sequentially linked structure instead.





- Although the interface (functions called) of lists and vectors and their iterators are quite similar, their implementations are VERY different. Clues to these differences can be seen in the operations that are NOT in common, such as:
  - STL `vector`s / arrays allow "random-access" / indexing / `[]` subscripting. We can immediately jump to an arbitrary location within the vector / array.
  - STL `list`s have no subscripting operation (we can't use `[]` to access data). The only way to get to the middle of a list is to follow pointers one link at a time.
  - Lists have `push_front` and `pop_front` functions in addition to the `push_back` and `pop_back` functions of vectors.
  - `erase` and `insert` in the middle of the STL `list` is very efficient, independent of the size of the list. Both are implemented by rearranging pointers between the small blocks of memory. (We'll see this when we discuss the implementation details next week).
  - We can't use the same STL `sort` function we used for `vector`; we must use a special `sort` function defined by the STL `list` type.

    ```
    std::vector<int> my_vec;
    std::list<int> my_lst;
    // ... put some data in my_vec & my_lst
    std::sort(my_vec.begin(),my_vec.end(),optional_compare_function);
    my_lst.sort(optional_compare_function);
    ```

    Note: STL `list sort` member function is just as efficient, $O(n \log n)$, and will also take the same optional compare function as STL `vector`.
  - Several operations invalidate the values of vector iterators, but not list iterators:
    - `erase` invalidates all iterators after the point of erasure in vectors;
    - `push_back` and `resize` invalidate ALL iterators in a vector

    The value of any associated vector iterator must be re-assigned / re-initialized after these operations.

9.10 Exercise: Revising the Class List Program to Use Lists (& Iterators)

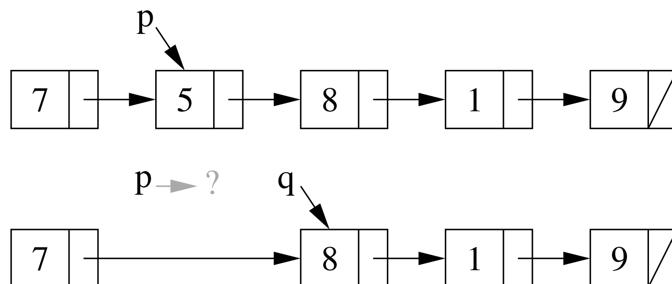
Now let's further modify the program to use lists instead of vectors. Because we've already switched to iterators, this change will be relatively easy. And now the program will be more efficient!

9.11 Erase & Iterators

- STL `list`s and `vector`s each have a special member function called `erase`. In particular, given list of ints `s`, consider the example:

  ```
  std::list<int>::iterator p = s.begin();
  ++p;
  std::list<int>::iterator q = s.erase(p);
  ```

- After the code above is executed:
  - The integer stored in the second entry of the list has been removed.
  - The size of the list has shrunk by one.
  - The iterator `p` does not refer to a valid entry.
  - The iterator `q` refers to the item that was the third entry and is now the second.



- To reuse the iterator `p` and make it a valid entry, you will often see the code written:

  ```
  std::list<int>::iterator p = s.begin();
  ++p;
  p = s.erase(p);
  ```

- Even though the `erase` function has the same syntax for vectors and for list, the vector version is $O(n)$, whereas the list version is $O(1)$.

9.12 Insert

- Similarly, there is an `insert` function for STL lists that takes an iterator and a value and adds a link in the chain with the new value immediately before the item pointed to by the iterator.
- The call returns an iterator that points to the newly added element. Variants on the basic insert function are also defined.

9.13 Exercise: Using STL `list` Erase & Insert

Write a function that takes an STL `list` of integers, `lst`, and an integer, `x`. The function should 1) remove all negative numbers from the list, 2) verify that the remaining elements in the list are sorted in increasing order, and 3) insert `x` into the list such that the order is maintained.

9.14 Implementing `Vec<T>` Iterators

- Let's add iterators to our `Vec<T>` class declaration from last lecture:

```
public:
  // TYPEDEFS
  typedef T* iterator;
  typedef const T* const_iterator;

  // MODIFIERS
  iterator erase(iterator p);

  // ITERATOR OPERATIONS
  iterator begin() { return m_data; }
  const_iterator begin() const { return m_data; }
  iterator end() { return m_data + m_size; }
  const_iterator end() const { return m_data + m_size; }
```

- First, remember that `typedef` statements create custom, alternate names for existing types.

  `Vec<int>::iterator` is an iterator type defined by the `Vec<int>` class. It is just a `T *` (an `int *`). Thus, internal to the declarations and member functions, `T*` and `iterator` **may be used interchangeably**.
- Because the underlying implementation of `Vec` uses an array, and because pointers *are* the "iterator"s of arrays, the implementation of vector iterators is quite simple. *Note: the implementation of iterators for other STL containers is more involved!*
- Thus, `begin()` returns a pointer to the first slot in the `m_data` array. And `end()` returns a pointer to the "slot" just beyond the last legal element in the `m_data` array (as prescribed in the STL standard).
- Furthermore, dereferencing a `Vec<T>::iterator` (dereferencing a pointer to type `T`) correctly returns one of the objects in the `m_data`, an object with type `T`.
- And similarly, the `++`, `--`, `<`, `==`, `!=`, `>=`, etc. operators on pointers automatically apply to `Vec` iterators.
- The `erase` function requires a bit more attention. We've implemented the core of this function above. The STL standard further specifies that the return value of `erase` is an iterator pointing to the new location of the element just after the one that was deleted.
- Finally, note that after a `push_back` or `erase` or `resize` call some or all iterators referring to elements in that vector may be *invalidated*. Why? You must take care when designing your program logic to avoid invalid iterator bugs!

CSCI-1200 Data Structures — Spring 2017
Lecture 10 — Vector Iterators & Linked Lists

Review from Lecture 9

- Explored a program to maintain a class enrollment list and an associated waiting list.
- Unfortunately, erasing items from the front or middle of vectors is inefficient.
- Iterators can be used to access elements of a vector
- Iterators and iterator operations (increment, decrement, erase, & insert)
- STL's `list` class
- Differences between indices and iterators, differences between STL `list` and STL `vector`.

Today's Class

- Quick review of iterators
- Implementation of iterators in our homemade `Vec` class (from Lecture 8)
- `const` and reference on return values
- Building *our own* basic linked lists:
  - Stepping through a list
  - Push back
  - ... & even more in the next couple lectures!

10.1 Review: Iterators and Iterator Operations

- An iterator type is defined by each STL container class. For example:

```
std::vector<double>::iterator v_itr;
std::list<std::string>::iterator l_itr;
std::string::iterator s_itr;
```

- An iterator is assigned to a specific location in a container. For example:

```
v_itr = vec.begin() + i;   //  i-th location in a vector
l_itr = lst.begin();       //  first entry in a list
s_itr = str.begin();       //  first char of a string
```

  Note: We can add an integer to vector and string iterators, but not to list iterators.

- The contents of the specific entry referred to by an iterator are accessed using the `*` *dereference operator*:
  In the first and third lines, `*v_itr` and `*l_itr` are l-values. In the second, `*s_itr` is an r-value.

```
*v_itr = 3.14;
cout << *s_itr << endl;
*l_itr = "Hello";
```

- Stepping through a container, either forward and backward, is done using increment (`++`) and decrement (`--`) operators:

```
++itr;   itr++;   --itr;   itr--;
```

  These operations move the iterator to the next and previous locations in the vector, list, or string. The operations do not change the contents of container!

- Finally, we can change the container that a specific iterator is attached to **as long as the types match**. Thus, if `v` and `w` are both `std::vector<double>`, then the code:

```
v_itr = v.begin();
*v_itr = 3.14;   // changes 1st entry in v
v_itr = w.begin() + 2;
*v_itr = 2.78;   // changes 3rd entry in w
```

  works fine because `v_itr` is a `std::vector<double>::iterator`, but if `a` is a `std::vector<std::string>` then

```
v_itr = a.begin();
```

  is a syntax error because of a type clash!

10.2 Additional Iterator Operations for Vector (& String) Iterators

- Initialization at a random spot in the vector:

  ```
  v_itr = v.begin() + i;
  ```

  Jumping around inside the vector through addition and subtraction of location counts:

  ```
  v_itr = v_itr + 5;
  ```

  moves `p` 5 locations further in the vector. These operations are constant time, $O(1)$ for vectors.

- These operations are *not allowed* for list iterators (and most other iterators, for that matter) because of the way the corresponding containers are built. These operations would be linear time, $O(n)$, for lists, where n is the number of slots jumped forward/backward. Thus, they are not provided by STL for lists.

- Students are often confused by the difference between iterators and indices for vectors. Consider the following declarations:

  ```
  std::vector<double> a(10, 2.5);
  std::vector<double>::iterator p = a.begin() + 5;
  unsigned int i=5;
  ```

- Iterator `p` refers to location 5 in vector `a`. The value stored there is directly accessed through the `*` operator:

  ```
  *p = 6.0;
  cout << *p << endl;
  ```

- The above code has **changed the contents** of vector `a`. Here's the equivalent code using subscripting:

  ```
  a[i] = 6.0;
  cout << a[i] << endl;
  ```

- Here's another common confusion:

```
std::list<int> lst;      lst.push_back(100); lst.push_back(200);
     lst.push_back(300); lst.push_back(400); lst.push_back(500);

std::list<int>::iterator itr,itr2,itr3;
itr = lst.begin();// itr is pointing at the 100
++itr;            // itr is now pointing at 200
*itr += 1;        // 200 becomes 201
// itr += 1;      // does not compile!  can't advance list iterator like this

itr = lst.end();  // itr is pointing "one past the last legal value" of lst
itr--;            // itr is now pointing at 500;
itr2 = itr--;     // itr is now pointing at 400, itr2 is still pointing at 500
itr3 = --itr;     // itr is now pointing at 300, itr3 is also pointing at 300

// dangerous: decrementing the begin iterator is "undefined behavior"
//  (similarly, incrementing the end iterator is also undefined)
//  it may seem to work, but break later on this machine or on another machine!
itr = lst.begin();
itr--;   // dangerous!
itr++;
assert (*itr == 100);  // might seem ok...  but rewrite the code to avoid this!
```

10.3 STL List: Erase (review) & Insert (skipped last time)

- The `erase` member function (for STL `vector` and STL `list`) takes in a single argument, an iterator pointing at an element in the container. It removes that item, and the function returns an iterator pointing at the element after the removed item.

- Similarly, there is an `insert` function for STL `vector` and STL `list` that takes in 2 arguments, an iterator and a new element, and adds that element immediately before the item pointed to by the iterator. The function returns an iterator pointing at the newly added element.

- Even though the `erase` and `insert` functions have the same syntax for vector and for list, the vector versions are $O(n)$, whereas the list versions are $O(1)$.

- Iterators positioned on an STL `vector`, at or after the point of an `erase` operation, are invalidated. Iterators positioned anywhere on an STL `vector` *may be* invalid after an `insert` (or `push_back` or `resize`) operation.
- Iterators attached to an STL `list` are not invalidated after an `insert` or `erase` (except iterators attached to the erased element!) or `push_back`/`push_front`.

10.4 Exercise: Using STL `list` Erase & Insert

Write a function that takes an STL `list` of integers, `lst`, and an integer, `x`. The function should 1) remove all negative numbers from the list, 2) verify that the remaining elements in the list are sorted in increasing order, and 3) insert `x` into the list such that the order is maintained.

10.5 Implementing `Vec<T>` Iterators

- Let's add iterators to our `Vec<T>` class declaration from Lecture 8:

```
public:
  // TYPEDEFS
  typedef T* iterator;
  typedef const T* const_iterator;

  // MODIFIERS
  iterator erase(iterator p);

  // ITERATOR OPERATIONS
  iterator begin() { return m_data; }
  const_iterator begin() const { return m_data; }
  iterator end() { return m_data + m_size; }
  const_iterator end() const { return m_data + m_size; }
```

- First, remember that `typedef` statements create custom, alternate names for existing types.

  `Vec<int>::iterator` is an iterator type defined by the `Vec<int>` class. It is just a `T *` (an `int *`). Thus, internal to the declarations and member functions, `T*` and `iterator` **may be used interchangeably**.
- Because the underlying implementation of `Vec` uses an array, and because pointers *are* the "iterator"s of arrays, the implementation of vector iterators is quite simple. *Note: the implementation of iterators for other STL containers is more involved! We'll see how STL* `list` *iterators work in a later lecture.*
- Thus, `begin()` returns a pointer to the first slot in the `m_data` array. And `end()` returns a pointer to the "slot" just beyond the last legal element in the `m_data` array (as prescribed in the STL standard).
- Furthermore, dereferencing a `Vec<T>::iterator` (dereferencing a pointer to type `T`) correctly returns one of the objects in the `m_data`, an object with type `T`.
- And similarly, the `++`, `--`, `<`, `==`, `!=`, `>=`, etc. operators on pointers automatically apply to `Vec` iterators. We don't need to write any additional functions for iterators, since we get all of the necessary behavior from the underlying pointer implementation.
- The `erase` function requires a bit more attention. We've implemented a version of this function in the previous lecture. The STL standard further specifies that the return value of `erase` is an iterator pointing to the new location of the element just after the one that was deleted.

10.6 References and Return Values

- A reference is an *alias* for another variable. For example:

```
string a = "Tommy";
string b = a;    // a new string is created using the string copy constructor
string& c = a;   // c is an alias/reference to the string object a
b[1] = 'i';
cout << a << " " << b << " " << c << endl;    // outputs:  Tommy Timmy Tommy
c[1] = 'a';
cout << a << " " << b << " " << c << endl;    // outputs:  Tammy Timmy Tammy
```

  The reference variable `c` refers to the same string as variable `a`. Therefore, when we change `c`, we change `a`.

- Exactly the same thing occurs with reference parameters to functions and the return values of functions. Let's look at the `Student` class from Lecture 4 again:

```
class Student {
public:
  const string& first_name() const { return first_name_; }
  const string& last_name() const { return last_name_; }
private:
  string first_name_;
  string last_name_;
};
```

- In the main function we had a vector of students:

```
vector<Student> students;
```

  Based on our discussion of references above and looking at the class declaration, what if we wrote the following. Would the code then be changing the internal contents of the i-th Student object?

```
string & fname = students[i].first_name();
fname[1] = 'i'
```

- The answer is NO! The `Student` class member function `first_name` returns a **const** reference. The compiler will complain that the above code is attempting to assign a const reference to a non-const reference variable.
- If we instead wrote the following, then compiler would complain that you are trying to change a const object.

```
const string & fname = students[i].first_name();
fname[1] = 'i'
```

- Hence in both cases the Student class would be "safe" from attempts at external modification.
- However, the author of the `Student` class would get into trouble if the member function return type was only a reference, and not a const reference. Then external users could access and change the internal contents of an object! This is a bad idea in most cases.

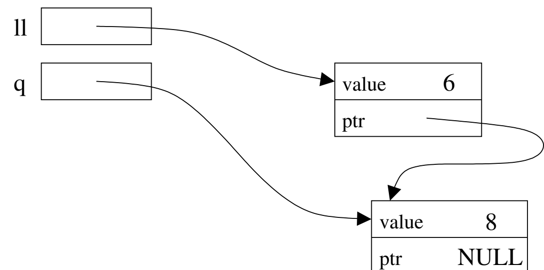
10.7 Working towards *our own* version of the STL `list`

- Our discussion of how the STL `list<T>` is implemented has been intuitive: it is a "chain" of objects.
- Now we will study the underlying mechanism — *linked lists*.
- This will allow us to build custom classes that mimic the STL `list` class, and add extensions and new features (more in the next couple lectures!).

10.8 Objects with Pointers, Linking Objects Together

- The two fundamental mechanisms of linked lists are:
  - creating objects with pointers as one of the member variables, and
  - making these pointers point to other objects of the same type.
- These mechanisms are illustrated in the following program:

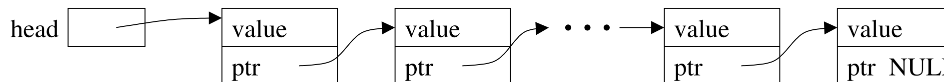
```
template <class T>
class Node {
public:
  T value;
  Node* ptr;
};

int main() {
  Node<int>* ll;       // ll is a pointer to a (non-existent) Node
  ll = new Node<int>;  // Create a Node and assign its memory address to ll
  ll->value = 6;       // This is the same as (*ll).value = 6;
  ll->ptr = NULL;      // NULL == 0, which indicates a "null" pointer

  Node<int>* q = new Node<int>;
  q->value = 8;
  q->ptr = NULL;

  // set ll's ptr member variable to
  // point to the same thing as variable q
  ll->ptr = q;

  cout << "1st value: " << ll->value << "\n"
       << "2nd value: " << ll->ptr->value << endl;
}
```



10.9 Definition: A Linked List

- The definition is recursive: A linked list is either:
  - Empty, or
  - Contains a node storing a value and a pointer to a linked list.
- The first node in the linked list is called the *head* node and the pointer to this node is called the *head* pointer. The pointer's value will be stored in a variable called `head`.

10.10 Visualizing Linked Lists



- The `head` pointer variable is drawn with its own box. It is an individual variable. It is important to have a separate pointer to the first node, since the "first" node may change.
- The objects (nodes) that have been dynamically allocated and stored in the linked lists are shown as boxes, with arrows drawn to represent pointers.
  - Note that this is a conceptual view only. The memory locations could be anywhere, and the actual values of the memory addresses aren't usually meaningful.
- The last node MUST have NULL for its pointer value — you will have all sorts of trouble if you don't ensure this!
- You should make a habit of drawing pictures of linked lists to figure out how to do the operations.

10.11 Basic Mechanisms: Stepping Through the List

- We'd like to write a function to determine if a particular value, stored in `x`, is also in the list.
- We can access the entire contents of the list, one step at a time, by starting just from the `head` pointer.
  - We will need a separate, local pointer variable to point to nodes in the list as we access them.
  - We will need a loop to step through the linked list (using the pointer variable) and a check on each value.

10.12 Exercise: Write `is_there`

```
template <class T> bool is_there(Node<T>* head, const T& x) {
```

- If the input linked list chain contains n elements, what is the order notation of `is_there`?

10.13 Basic Mechanisms: Pushing on the Back

- Goal: place a new node at the end of the list.
- We must step to the end of the linked list, remembering the pointer to the last node.
  - This is an $O(n)$ operation and is a major drawback to the ordinary linked-list data structure we are discussing now. We will correct this drawback by creating a slightly more complicated linking structure in our next lecture.
- We must create a new node and attach it to the end.
- We must remember to update the `head` pointer variable's value if the linked list is initially empty.
  - Hence, in writing the function, we must pass the pointer variable **by reference**.

10.14 Exercise: Write `push_front`

```
template <class T> void push_front( Node<T>* & head, T const& value ) {
```

- If the input linked list chain contains n elements, what is the order notation of the implementation of `push_front`?

10.15 Exercise: Write `push_back`

```
template <class T> void push_back( Node<T>* & head, T const& value ) {
```

- If the input linked list chain contains n elements, what is the order notation of this implementation of `push_back`?

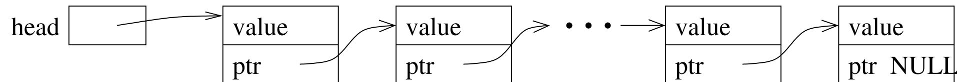
10.16 Next time... Can we get better performance out of linked lists? Yes!

CSCI-1200 Data Structures — Spring 2017
Lectures 11 — Doubly Linked Lists

Review from Lecture 10

- Review of iterators, implementation of iterators in our homemade `Vec` class
- `const` and reference on return values
- Building *our own* basic linked lists: Stepping through a list & push back

```
template <class T>
class Node {
public:
  T value;
  Node* ptr;
};
```



  - Stepping through a list

```
template <class T>
bool is_there(Node<T>* head, const T& x) {
  for (Node<T> *p  = head; p != NULL ; p = p->ptr) {
    if (p->value == x) return true;
  }
  return false;
}
```

Today's Lecture

- STL List w/ iterators vs. "homemade" linked list with Node objects & pointers
- Basic linked list operations, continued: Insert & Remove
- Common mistakes
- Limitations of singly-linked lists
- Doubly-linked lists:
  - Structure
  - Insert
  - Remove

11.1 Basic Mechanisms: Inserting a Node

- There are two parts to this: finding the location where the insert must take place, and doing the insert operation.
- We will ignore the find for now. We will also write only a code segment to understand the mechanism rather than writing a complete function.
- The insert operation itself requires that we have a pointer to the location **before** the insert location.
- If `p` is a pointer to this node, and `x` holds the value to be inserted, then the following code will do the insertion. Draw a picture to illustrate what is happening.

```
Node<T> * q = new Node<T>; //  create a new node
q -> value = x;            //  store x in this node
q -> next = p -> next;     //  make its successor be the current successor of p
p -> next = q;             //  make p's successor be this new node
```

- Note: This code will not work if you want to insert `x` in a new node at the *front* of the linked list. Why not?

11.2 Basic Mechanisms: Removing a Node

- There are two parts to this: finding the node to be removed and doing the remove operation.
- The remove operation itself requires a pointer to the node **before** the node to be removed.
- Removing the first node is an important special case.

11.3 Exercise: Remove a Node

Suppose `p` points to a node that should be removed from a linked list, `q` points to the node before `p`, and `head` points to the first node in the linked list. Write code to remove `p`, making sure that if `p` points to the first node that `head` points to what was the second node and now is the first after `p` is removed. Draw a picture of each scenario.

11.4 Exercise: List Copy

Write a *recursive* function to copy all nodes in a linked list to form an new linked list of nodes with identical structure and values. Here's the function prototype:

```
template <class T> void CopyAll(Node<T>* old_head,  Node<T>*& new_head) {
```

11.5 Exercise: Remove All

Write a *recursive* function to delete all nodes in a linked list. Here's the function prototype:

```
template <class T> void RemoveAll(Node<T>*& head) {
```

11.6 Basic Linked Lists Mechanisms: Common Mistakes

Here is a summary of common mistakes. Read these carefully, and read them again when you have a problem that you need to solve.

- Allocating a new node to step through the linked list; only a pointer variable is needed.
- Confusing the `.` and the `->` operators.
- Not setting the pointer from the last node to NULL.
- Not considering special cases of inserting / removing at the beginning or the end of the linked list.
- Applying the `delete` operator to a node (calling the operator on a pointer to the node) before it is appropriately disconnected from the list. Delete should be done after all pointer manipulations are completed.
- Pointer manipulations that are out of order. These can ruin the structure of the linked list.
- Trying to use STL iterators to visit elements of a "home made" linked list chain of nodes. (And the reverse.... trying to use `->next` and `->prev` with STL list iterators.)

11.7 Limitations of Singly-Linked Lists

- We can only move through it in one direction
- We need a pointer to the node **before** the node that needs to be deleted.
- Appending a value at the end requires that we step through the entire list to reach the end.

11.8 Generalizations of Singly-Linked Lists

- Three common generalizations:
  - Doubly-linked: allows forward and backward movement through the nodes
  - Circularly linked: simplifies access to the tail, when doubly-linked
  - Dummy header node: simplifies special-case checks
- Today we will explore and implement a doubly-linked structure.

11.9 Transition to a doubly-linked list

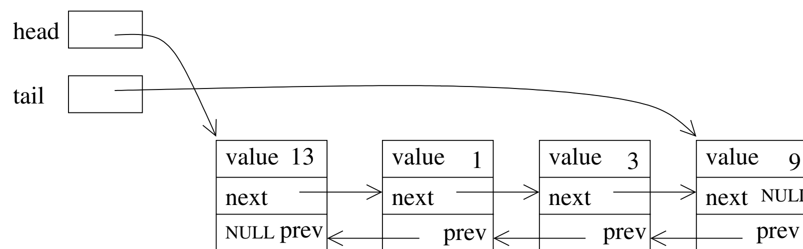
- The revised `Node` class has two pointers, one going "forward" to the successor in the linked list and one going "backward" to the predecessor in the linked list. We will have a `head` pointer to the beginning *and* a `tail` pointer to the end of the list.

```
template <class T> class Node {
public:
    Node() : next_(NULL), prev_(NULL) {}
    Node(const T& v) : value_(v), next_(NULL), prev_(NULL) {}
    T value_;
    Node<T>* next_;
    Node<T>* prev_;
};
```

- First we'll reimplement some of the basic mechanisms we've already worked through for singly-linked lists. In the next lecture we'll build the full `ds_list` class and will define the list iterators as a class inside a class.

11.10 The Structure of Doubly-Linked Lists

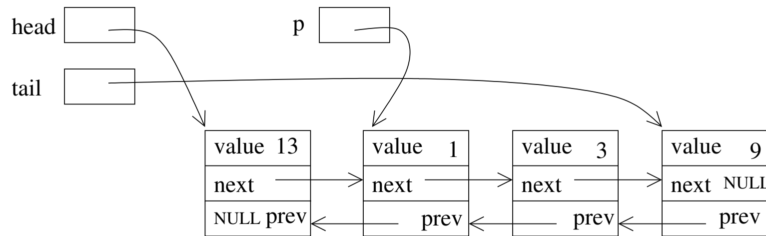
- Here is a picture of a doubly-linked list holding four integer values:



- Note that we now assume that we have both a `head` pointer, as before and a `tail` pointer variable, which stores the address of the last node in the linked list.
- The tail pointer is not strictly necessary, but it allows immediate access to the end of the list for efficient push-back operations.

11.11 Inserting in the Middle of a Doubly-Linked List

- Suppose we want to insert a new node containing the value 15 following the node containing the value 1. We have a temporary pointer variable, `p`, that stores the address of the node containing the value 1. Here's a picture of the state of affairs:



- What must happen?
  - The new node must be created, using another temporary pointer variable to hold its address.
  - Its two pointers must be assigned.
  - Two pointers in the current linked list must be adjusted. Which ones?

  Assigning the pointers for the new node MUST occur before changing the pointers for the current linked list nodes!

- At this point, we are ignoring the possibility that the linked list is empty or that `p` points to the tail node (`p` pointing to the head node doesn't cause any problems).
- **Exercise:** write the code as just described.

11.12 Removing from the Middle of a Doubly-Linked List

- Suppose now instead of inserting a value we want to remove the node pointed to by `p` (the node whose address is stored in the pointer variable `p`)
- Two pointers need to change before the node is deleted! All of them can be accessed through the pointer variable `p`.
- **Exercise:** write this code.

11.13 Special Cases of Remove

- If `p==head` and `p==tail`, the single node in the list must be removed and both the `head` and `tail` pointer variables must be assigned the value `NULL`.
- If `p==head` or `p==tail`, then the pointer adjustment code we just wrote needs to be specialized to removing the first or last node.
- Next lecture we'll write the `erase` function as part of our implementation mimicing the STL `list` class.

CSCI-1200 Data Structures — Spring 2017
Lecture 12 — List Implementation

- Exam 2 will be Monday evening March 6th from 6-8pm. Practice problems are available on the calendar.
- Your exam room & zone assignment will be posted on the homework submission site by the end of the week. *Note: We are re-shuffling the room & zone assignments from Exam 1.*

Review from Lecture 11

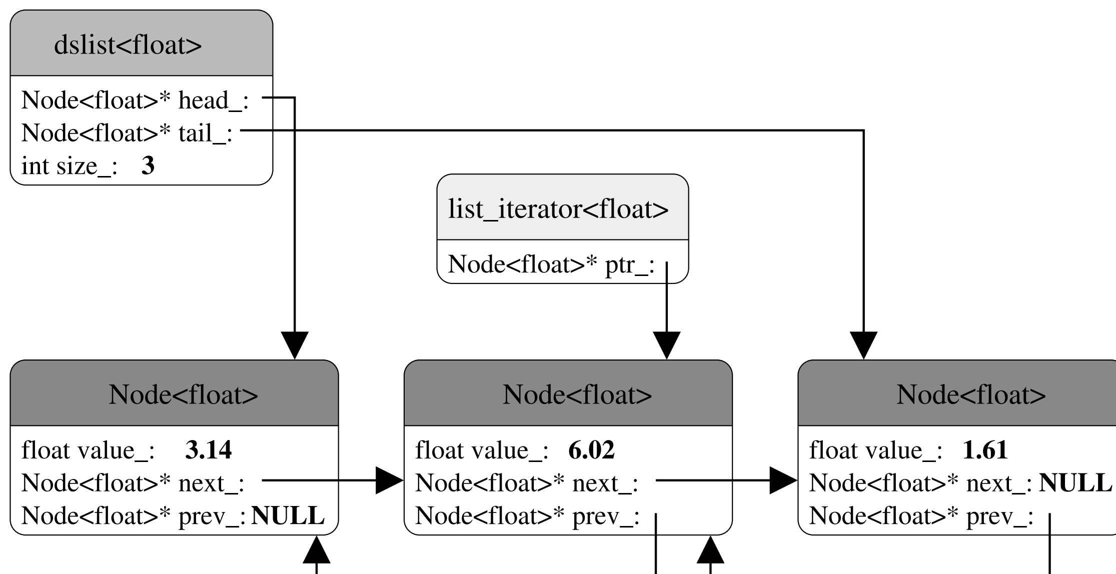
- Limitations of singly-linked lists
- Doubly-linked lists: Structure, Insert, & Remove
  - Note: We didn't finish all of the special/corner cases for remove from a doubly-linked list. Does it matter? Story time....

Today's Lecture

- Our own version of the STL `list<T>` class, named `dslist`
- Implementing list iterators

12.1 The dslist Class — Overview

- We will write a templated class called `dslist` that implements much of the functionality of the `std::list<T>` container and uses a doubly-linked list as its internal, low-level data structure.
- Three classes are involved: the node class, the iterator class, and the dslist class itself.
- Below is a basic diagram showing how these three classes are related to each other:



- For each list object created by a program, we have one instance of the `dslist` class, and multiple instances of the `Node`. For each iterator variable (of type `dslist<T>::iterator`) that is used in the program, we create an instance of the `list_iterator` class.

12.2 The Node Class

- It is ok to make all members public because individual nodes are never seen outside the list class. (`Node` objects are not accessible to a user through the public `dslist` interface.)
- Another option to ensure the `Node` member variables stay private would be to nest the entire `Node` class inside of the private section of the `dslist` declaration. We'll see an example of this later in the term.
- Note that the constructors initialize the pointers to NULL.

12.3 The Iterator Class — Desired Functionality

- Increment and decrement operators (operations that follow links through pointers).
- Dereferencing to access contents of a node in a list.
- Two comparison operations: `operator==` and `operator!=`.

12.4 The Iterator Class — Implementation

- Separate class.
- Stores a pointer to a node in a linked list.
- Constructors initialize the pointer — they will be called from the `dslist<T>` class member functions.
  - `dslist<T>` is a friend class to allow access to the iterators `ptr_` pointer variable (needed by `dslist<T>` member functions such as `erase` and `insert`).
- `operator*` dereferences the pointer and gives access to the contents of a node. (The user of a `dslist` class is never given full access to a `Node` object!)
- Stepping through the chain of the linked-list is implemented by the increment and decrement operators.
- `operator==` and `operator!=` are defined, but no other comparison operators are allowed.

12.5 The dslist Class — Overview

- Manages the actions of the iterator and node classes.
- Maintains the head and tail pointers and the size of the list. (member variables: `head_`, `tail_`, `size_`)
- Manages the overall structure of the class through member functions.
- Typedef for the `iterator` name.
- Prototypes for member functions, which are equivalent to the `std::list<T>` member functions.
- Some things are missing, most notably `const_iterator` and `reverse_iterator`.

12.6 The dslist class — Implementation Details

- Many short functions are in-lined
- Clearly, it must contain the "big 3": copy constructor, `operator=`, and destructor. The details of these are realized through the private `copy_list` and `destroy_list` member functions.

12.7 C++ Template Implementation Detail - Using `typename`

- The use of typedefs within a templated class, for example the `dslist<T>::iterator` can confuse the compiler because it is a *template-parameter dependent name* and is thus ambiguous in some contexts. (Is it a value or is it a type?)
- If you get a strange error during compilation (where the compiler is clearly confused about seemingly clear and logical code), you will need to explicitly let the compiler know that it is a type by putting the `typename` keyword in front of the type. For example, inside of the `operator==` function:

```
typename dslist<T>::iterator left_itr = left.begin();
```

- Don't worry, we'll never test you on where this keyword is needed. Just be prepared to use it when working on the homework.

12.8 Exercises

1. Write `dslist<T>::push_front`
2. Write `dslist<T>::erase`

```
#ifndef dslist_h_
#define dslist_h_
// A simplified implementation of a generic list container class,
// including the iterator, but not the const_iterators.  Three
// separate classes are defined: a Node class, an iterator class, and
// the actual list class.  The underlying list is doubly-linked, but
// there is no dummy head node and the list is not circular.
#include <cassert>

// -----------------------------------------------------------------
// NODE CLASS
template <class T>
class Node {
public:
  Node() : next_(NULL), prev_(NULL) {}
  Node(const T& v) : value_(v), next_(NULL), prev_(NULL) {}

  // REPRESENTATION
  T value_;
  Node<T>* next_;
  Node<T>* prev_;
};

// A "forward declaration" of this class is needed
template <class T> class dslist;

// -----------------------------------------------------------------
// LIST ITERATOR
template <class T>
class list_iterator {
public:
  // default constructor, copy constructor, assignment operator, & destructor
  list_iterator() : ptr_(NULL) {}
  list_iterator(Node<T>* p) : ptr_(p) {}
  list_iterator(const list_iterator<T>& old) : ptr_(old.ptr_) {}
  list_iterator<T>& operator=(const list_iterator<T>& old) {
    ptr_ = old.ptr_;  return *this; }
  ~list_iterator() {}

  // dereferencing operator gives access to the value at the pointer
  T& operator*()  { return ptr_->value_;  }

  // increment & decrement operators
  list_iterator<T>& operator++() { // pre-increment, e.g., ++iter
    ptr_ = ptr_->next_;
    return *this;
  }
  list_iterator<T> operator++(int) { // post-increment, e.g., iter++
    list_iterator<T> temp(*this);
    ptr_ = ptr_->next_;
    return temp;
  }
  list_iterator<T>& operator--() { // pre-decrement, e.g., --iter
    ptr_ = ptr_->prev_;
    return *this;
  }
  list_iterator<T> operator--(int) { // post-decrement, e.g., iter--
    list_iterator<T> temp(*this);
    ptr_ = ptr_->prev_;
    return temp;
  }

  // the dslist class needs access to the private ptr_ member variable
  friend class dslist<T>;

  // Comparions operators are straightforward
  bool operator==(const list_iterator<T>& r) const {
    return ptr_ == r.ptr_; }
  bool operator!=(const list_iterator<T>& r) const {
    return ptr_ != r.ptr_; }

private:
  // REPRESENTATION
  Node<T>* ptr_;    // ptr to node in the list

};

// -----------------------------------------------------------------
// LIST CLASS DECLARATION
// Note that it explicitly maintains the size of the list.
template <class T>
class dslist {
public:
  // default constructor, copy constructor, assignment operator, & destructor
  dslist() : head_(NULL), tail_(NULL), size_(0) {}
  dslist(const dslist<T>& old) { this->copy_list(old); }
  dslist& operator= (const dslist<T>& old);
  ~dslist() { this->destroy_list(); }

  // simple accessors & modifiers
  unsigned int size() const { return size_; }
  bool empty() const { return head_ == NULL; }
  void clear() { this->destroy_list(); }

  // read/write access to contents
  const T& front() const { return head_->value_;  }
  T& front() { return head_->value_; }
  const T& back() const { return tail_->value_; }
  T& back() { return tail_->value_; }

  // modify the linked list structure
  void push_front(const T& v);
  void pop_front();
  void push_back(const T& v);
  void pop_back();

  typedef list_iterator<T> iterator;
  iterator erase(iterator itr);
  iterator insert(iterator itr, const T& v);
  iterator begin() { return iterator(head_); }
  iterator end() { return iterator(NULL); }

private:
  // private helper functions
  void copy_list(const dslist<T>& old);
  void destroy_list();

  //REPRESENTATION
  Node<T>* head_;
  Node<T>* tail_;
  unsigned int size_;
};
```

```cpp
// -------------------------------------------------------------------
// LIST CLASS IMPLEMENTATION
template <class T>
dslist<T>& dslist<T>::operator= (const dslist<T>& old) {
  // check for self-assignment
  if (&old != this) {
    this->destroy_list();
    this->copy_list(old);
  }
  return *this;
}

template <class T>
void dslist<T>::push_front(const T& v) {

}

template <class T>
void dslist<T>::pop_front() {

}

template <class T>
void dslist<T>::push_back(const T& v) {

}

template <class T>
void dslist<T>::pop_back() {

}

// do these lists look the same (length & contents)?
template <class T>
bool operator== (dslist<T>& left, dslist<T>& right) {
  if (left.size() != right.size()) return false;
  typename dslist<T>::iterator left_itr = left.begin();
  typename dslist<T>::iterator right_itr = right.begin();
  // walk over both lists, looking for a mismatched value
  while (left_itr != left.end()) {
    if (*left_itr != *right_itr) return false;
    left_itr++; right_itr++;
  }
  return true;
}
template <class T>
bool operator!= (dslist<T>& left, dslist<T>& right){ return !(left==right); }

template <class T>
typename dslist<T>::iterator dslist<T>::erase(iterator itr) {

}

template <class T>
typename dslist<T>::iterator dslist<T>::insert(iterator itr, const T& v) {

}

template <class T>
void dslist<T>::copy_list(const dslist<T>& old) {

}

template <class T>
void dslist<T>::destroy_list() {

}

#endif
```

CSCI-1200 Data Structures — Spring 2017
Lecture 13 — Advanced Recursion

Announcements: Test 2 Information

- Test 2 will be held **Monday, Mar. 6th** from 6-8pm.
  Your test room & zone assignment is posted on the homework submission site.
  Note: We have re-shuffled the room & zone assignments from Test 1.
- No make-ups will be given except for emergency situations, and even then a written excuse from the Dean of Students or the Office of Student Experience will be required.
- Coverage: Lectures 1-13, Labs 1-7, HW 1-5.
- Closed-book and closed-notes *except for 1 sheet of notes on 8.5x11 inch paper (front & back) that may be handwritten or printed*. Computers, cell-phones, palm pilots, calculators, PDAs, music players, etc. are not permitted and must be turned off.
- All students must bring their Rensselaer photo ID card.
- Practice problems from previous tests are available on the course website. Solutions to the problems will be posted on Friday afternoon.

Test Taking Skills

- Look at the point values for each problem, allocate time proportional to the problem points. (Don't spend all of your time on one problem and neglect other big point problems).
- Look at the size of the answer box & the sample solution code line estimate for each problem. If your solution is going to take a lot more space than the box allows, we are probably looking for the solution to a simpler problem or a simpler solution to the problem.
- Going in to the test, you should know what big topics will be covered on the test. As you skim through the problems, see if you can match up those big topics to each question. Even if you are stumped about how to solve the whole problem, or some of the details of the problem, make sure you demonstrate your understanding of the big topic that is covered in that question.
- Re-read the problem statement carefully. Make sure you didn't miss anything.
- Ask questions during the test if something is unclear.

Review from Lecture 11 & Lab 7

- Limitations of singly-linked lists
- Doubly-linked lists:
  - Structure
  - Insert
  - Remove
- Our own version of the STL `list<T>` class, named `dslist`
- Implementing `list<T>::iterator`
- Importance of destructors & using Dr. Memory / Valgrind to find memory errors
- Decrementing the `end()` iterator

Today's Lecture

- Review Recursion vs. Iteration
  - Binary Search
- "Rules" for writing recursive functions
- Advanced Recursion — problems that cannot be easily solved using iteration (for or while loops):
  - Merge sort
  - Non-linear maze search

13.1 Review: Iteration vs. Recursion

- Every* recursive function can also be written iteratively. Sometimes the rewrite is quite simple and straightforward. Sometimes it's more work.
- Often writing recursive functions is more natural than writing iterative functions, especially for a first draft of a problem implementation.
- You should learn how to recognize whether an implementation is recursive or iterative, and practice rewriting one version as the other.
- Note: The order notation for the number of operations for the recursive and iterative versions of an algorithm is usually the same. However in C, C++, Java, and some other languages, *iterative functions are generally faster than their corresponding recursive functions*. This is due to the overhead of the function call mechanism. Compiler optimizations will sometimes (but not always!) reduce the performance hit by automatically eliminating the recursive function calls. This is called *tail call optimization*.

13.2 Binary Search

- Suppose you have a `std::vector<T> v` (for a placeholder type T), sorted so that:

```
v[0] <= v[1] <= v[2] <= ...
```

- Now suppose that you want to find if a particular value `x` is in the vector somewhere. How can you do this without looking at every value in the vector?
- The solution is a recursive algorithm called *binary search*, based on the idea of checking the middle item of the search interval within the vector and then looking either in the lower half or the upper half of the vector, depending on the result of the comparison.

```
template <class T>
bool binsearch(const std::vector<T> &v, int low, int high, const T &x) {
 if (high == low) return x == v[low];
  int mid = (low+high) / 2;
  if (x <= v[mid])
    return binsearch(v, low, mid, x);
  else
    return binsearch(v, mid+1, high, x);
}
template <class T>
bool binsearch(const std::vector<T> &v, const T &x) {
  return binsearch(v, 0, v.size()-1, x);
}
```

13.3 Exercises

1. What is the order notation of binary search?

2. Write a non-recursive version of binary search.

3. If we replaced the if-else structure inside the recursive binsearch function (above) with

```
if ( x < v[mid] )
  return binsearch( v, low, mid-1, x );
else
  return binsearch( v, mid, high, x );
```

   would the function still work correctly?

13.4 Rules for Writing Recursive Functions

Here is an outline of five steps that are useful in writing and debugging recursive functions. Note: You don't have to do them in exactly this order...

1. Handle the base case(s).
2. Define the problem solution in terms of smaller instances of the problem. Use *wishful thinking*, i.e., if someone else solves the problem of `fact(4)` I can extend that solution to solve `fact(5)`. This defines the necessary recursive calls. It is also the hardest part!
3. Figure out what work needs to be done before making the recursive call(s).
4. Figure out what work needs to be done after the recursive call(s) complete(s) to finish the computation. (What are you going to do with the result of the recursive call?)
5. Assume the recursive calls work correctly, but make sure they are progressing toward the base case(s)!

13.5 Another Recursion Example: Merge Sort

- Idea: 1) Split a vector in half, 2) Recursively sort each half, and 3) Merge the two sorted halves into a single sorted vector.
- Suppose we have a vector called `values` having two halves that are each already sorted. In particular, the values in subscript ranges `[low..mid]` (the lower interval) and `[mid+1..high]` (the upper interval) are each in increasing order.
- Which values are candidates to be the first in the final sorted vector? Which values are candidates to be the second?
- In a loop, the merging algorithm repeatedly chooses one value to copy to `scratch`. At each step, there are only two possibilities: the first uncopied value from the lower interval and the first uncopied value from the upper interval.
- The copying ends when one of the two intervals is exhausted. Then the remainder of the other interval is copied into the scratch vector. Finally, the entire scratch vector is copied back.

13.6 Exercise: Complete the Merge Sort Implementation

```
// prototypes
template <class T> void mergesort(std::vector<T>& values);
template <class T> void mergesort(int low, int high, std::vector<T>& values, std::vector<T>& scratch);
template <class T> void merge(int low, int mid, int high, std::vector<T>& values, std::vector<T>& scratch);

int main() {
  std::vector<double> pts(7);
  pts[0] = -45.0;  pts[1] =   89.0;  pts[2] =    34.7;  pts[3] =   21.1;
  pts[4] =   5.0;  pts[5] =  -19.0;  pts[6] =  -100.3;
  mergesort(pts);
  for (unsigned int i=0; i<pts.size(); ++i)
    std::cout << i << ": " << pts[i] << std::endl;
}

// The driver function for mergesort.  It defines a scratch std::vector for temporary copies.
template <class T> void mergesort(std::vector<T>& values) {
  std::vector<T> scratch(values.size());
  mergesort(0, int(values.size()-1), values, scratch);
}

// Here's the actual merge sort function.  It splits the std::vector in
// half, recursively sorts each half, and then merges the two sorted
// halves into a single sorted interval.
template <class T> void mergesort(int low, int high, std::vector<T>& values, std::vector<T>& scratch) {
  std::cout << "mergesort:  low = " << low << ", high = " << high << std::endl;
  if (low >= high)  //  intervals of size 0 or 1 are already sorted!
    return;
  int  mid = (low + high) / 2;
```

```
  mergesort(low, mid, values, scratch);
  mergesort(mid+1, high, values, scratch);
  merge(low, mid, high, values, scratch);
}

// Non-recursive function to merge two sorted intervals (low..mid & mid+1..high)
// of a std::vector, using "scratch" as temporary copying space.
template <class T>  void merge(int low, int mid, int high, std::vector<T>& values, std::vector<T>& scratch) {
  std::cout << "merge:  low = " << low << ", mid = " << mid << ", high = " << high << std::endl;
  int i=low, j=mid+1, k=low;

}
```

13.7 Thinking About Merge Sort

- It exploits the power of recursion! We only need to think about
  - Base case (intervals of size 1)
  - Splitting the vector
  - Merging the results
- We can insert `cout` statements into the algorithm and use this to understand how this is is happening.
- Can we analyze this algorithm and determine the order notation for the number of operations it will perform? Count the number of pairwise comparisons that are required.

13.8 Example: Word Search

- Take a look at the following grid of characters.

```
heanfuyaadfj
crarneradfad
chenenssartr
kdfthileerdr
chadufjavcze
dfhoepradlfc
neicpemrtlkf
paermerohtrr
diofetaycrhg
daldruetryrt
```

- The usual problem associated with a grid like this is to find words going forward, backward, up, down, or along a diagonal. Can you find "`computer`"?
- A sketch of the solution is as follows:
  - The grid of letters is represented as `vector<string> grid;` Each string represents a row. We can treat this as a *two-dimensional array*.
  - A word to be sought, such as "`computer`" is read as a string.
  - A pair of nested for loops searches the grid for occurrences of the first letter in the string. Call such a location (r, c)

  - At each such location, the occurrences of the second letter are sought in the 8 locations surrounding (r, c).
  - At each location where the second letter is found, a search is initiated in the direction indicated. For example, if the second letter is at $(r, c - 1)$, the search for the remaining letters proceeds up the grid.

- The implementation takes a bit of work, but is not too bad.

13.9 Example: Nonlinear Word Search

- Today we'll work on a different, but somewhat harder problem: What happens when we no longer require the locations to be along the same row, column or diagonal of the grid, but instead allow the locations to snake through the grid? The only requirements are that
  1. the locations of adjacent letters are connected along the same row, column or diagonal, and
  2. a location can not be used more than once in each word
- Can you find `rensselaer`? It is there. How about `temperature`? Close, but nope!
- The implementation of this is very similar to the implementation described above until after the first letter of a word is found.
- We will look at the code during lecture, and then consider how to write the recursive function.

13.10 Exercise: Complete the implementation

```
// Simple class to record the grid location.
class loc {
public:
  loc(int r=0, int c=0) : row(r), col(c) {}
  int row, col;
};
bool operator== (const loc& lhs, const loc& rhs) {
  return lhs.row == rhs.row && lhs.col == rhs.col;
}
// helper function to check if a position has already been used for this word
bool on_path(loc position, std::vector<loc> const& path) {
  for (unsigned int i=0; i<path.size(); ++i)
    if (position == path[i]) return true;
  return false;
}

bool search_from_loc(loc position /* current position */,
                     const std::vector<std::string>& board, const std::string& word,
                     std::vector<loc>& path /* path leading to the current pos */ ) {






}
```

```
// Read in the letter grid, the words to search and print the results
int main(int argc, char* argv[]) {
  if (argc != 2) {
    std::cerr << "Usage: " << argv[0] << " grid-file\n";
    return 1;
  }
  std::ifstream istr(argv[1]);
  if (!istr) {
    std::cerr << "Couldn't open " << argv[1] << '\n';
    return 1;
  }
  std::vector<std::string> board;
  std::string word;
  std::vector<loc> path;          //  The sequence of locations...
  std::string line;
  //  Input of grid from a file.  Stops when character '-' is reached.
  while ((istr  >> line) && line[0] != '-')
    board.push_back(line);
  while (istr >> word) {
    bool found = false;
    std::vector<loc> path;  //  Path of locations in finding the word
    //  Check all grid locations.  For any that have the first
    //  letter of the word, call the function search_from_loc
    //  to check if the rest of the word is there.
    for (unsigned int r=0; r<board.size() && !found; ++r) {
      for (unsigned int c=0; c<board[r].size() && !found; ++c) {
        if (board[r][c] == word[0] &&
            search_from_loc(loc(r,c), board, word, path))
          found = true;
      }
    }
    //  Output results
    std::cout << "\n** " << word << " ** ";
    if (found) {
      std::cout << "was found.  The path is \n";
      for(unsigned int i=0; i<path.size(); ++i)
        std::cout << "  " << word[i] << ":  (" << path[i].row << "," << path[i].col << ")\n";
    } else {
      std::cout << " was not found\n";
    }
  }
  return 0;
}
```

13.11 Summary of Nonlinear Word Search Recursion

- Recursion starts at each location where the first letter is found
- Each recursive call attempts to find the next letter by searching around the current position. When it is found, a recursive call is made.
- The current path is maintained at all steps of the recursion.
- The "base case" occurs when the path is full **or** all positions around the current position have been tried.

13.12 Exercise: Analyzing our Nonlinear Word Search Algorithm

- What is the order notation for the number of operations?

Final Note

We've said that recursion is sometimes the *most natural way* to begin thinking about designing and implementing many algorithms. It's ok if this feels downright uncomfortable right now. Practice, practice, practice!

CSCI-1200 Data Structures — Spring 2017
Lecture 14 — Problem Solving Techniques

Review from Lecture 13

- Rules for writing recursive functions:
  1. Handle the base case(s).
  2. Define the problem solution in terms of smaller instances of the problem. Use *wishful thinking*, i.e., if someone else solves the problem of `fact(4)` I can extend that solution to solve `fact(5)`. This defines the necessary recursive calls. It is also the hardest part!
  3. Figure out what work needs to be done before making the recursive call(s).
  4. Figure out what work needs to be done after the recursive call(s) complete(s) to finish the computation. (What are you going to do with the result of the recursive call?)
  5. Assume the recursive calls work correctly, but make sure they are progressing toward the base case(s)!
- Merge sort
- Non-linear maze search

Today's Class

- Today we will discuss how to design and implement algorithms using three steps or stages:
  1. Generating and Evaluating Ideas
  2. Mapping Ideas into Code
  3. Getting the Details Right

14.1 Generating and Evaluating Ideas

- Most importantly, play with examples! Can you develop a strategy for solving the problem? You should try any strategy on several examples. Is it possible to map this strategy into an algorithm and then code?
- Try solving a simpler version of the problem first and either learn from the exercise or generalize the result.
- Does this problem look like another problem you know how to solve?
- If someone gave you a partial solution, could you extend this to a complete solution?
- What if you split the problem in half and solved each half (recursively) separately?
- Does sorting the data help?
- Can you split the problem into different cases, and handle each case separately?
- Can you discover something fundamental about the problem that makes it easier to solve or makes you able to solve it more efficiently?
- Once you have an idea that you think will work, you should evaluate it: will it indeed work? are there other ways to approach it that might be better / faster? if it doesn't work, why not?

14.2 Mapping Ideas Into Code

- How are you going to represent the data? What is most efficient and what is easiest?
- Can you use classes to organize the data? What data should be stored and manipulated as a unit? What information needs to be stored for each object? What operations (beyond simple accessors) might be helpful?
- How can you divide the problem into units of logic that will become functions? Can you reuse any code you're previously written? Will any of the logic you write now be re-usable?
- Are you going to use recursion or iteration? What information do you need to maintain during the loops or recursive calls and how is it being "carried along"?
- How effective is your solution? Is your solution general? How is the performance? (What is the order notation of the number of operations)? Can you now think of better ideas or approaches?
- Make notes for yourself about the logic of your code as you write it. These will become your *invariants*; that is, what should be true at the beginning and end of each iteration / recursive call.

14.3 Getting the Details Right

- Is everything being initialized correctly, including boolean flag variables, accumulation variables, max / min variables?
- Is the logic of your conditionals correct? Check several times and test examples by hand.
- Do you have the bounds on the loops correct? Should you end at n, $n-1$ or $n-2$?
- Tidy up your "notes" to formalize the invariants. Study the code to make sure that your code does in fact have it right. When possible use assertions to test your invariants. (Remember, sometimes checking the invariant is impossible or too costly to be practical.)
- Does it work on the corner cases; e.g., when the answer is on the start or end of the data, when there are repeated values in the data, or when the data set is very small or very large?

14.4 Exercises: Practice using these Techniques on Simple Problems

- A perfect number is a number that is the sum of its factors. The first perfect number is 6. Let's write a program that finds all perfect numbers less than some input number n.

```
int main() {
  std::cout << "Enter a number: ";
  int n;
  std::cin >> n;
```

- Given a sequence of n floating point numbers, find the two that are closest in value.

```
int main() {

  float f;
  while (std::cin >> f) {

  }
```

- Now let's write code to remove duplicates from a sequence of numbers:

```
int main() {

  int x;
  while (std::cin >> x) {

  }
```

14.5 Example: Merge Sort

- In Lecture 13, we saw the basic framework for the merge sort algorithm and we finished the implementation of the merge helper function. How did we **Map Ideas Into Code**?
- What invariants can we write down within the `merge_sort` and `merge` functions? Which invariants can we test using assertions? Which ones are too expensive (i.e., will affect the overall performance of the algorithm)?

```
// We split the vector in half, recursively sort each half, and
// merge the two sorted halves into a single sorted interval.
template <class T>
void mergesort(int low, int high, vector<T>& values, vector<T>& scratch) {
  if (low >= high) return;
  int  mid = (low + high) / 2;

  mergesort(low, mid, values, scratch);
  mergesort(mid+1, high, values, scratch);

  merge(low, mid, high, values, scratch);

}


// Non-recursive function to merge two sorted intervals (low..mid & mid+1..high)
// of a vector, using "scratch" as temporary copying space.
template <class T>
void merge(int low, int mid, int high, vector<T>& values, vector<T>& scratch) {
  int i=low, j=mid+1, k=low;

  // while there's still something left in one of the sorted subintervals...
  while (i <= mid && j <= high) {

    // look at the top values, grab the smaller one, store it in the scratch vector
    if (values[i] < values[j]) {
      scratch[k] = values[i]; ++i;
    } else {
      scratch[k] = values[j]; ++j;
    }
    ++k;
  }

  // Copy the remainder of the interval that hasn't been exhausted
  for ( ; i<=mid; ++i, ++k ) scratch[k] = values[i];  // low interval
  for ( ; j<=high; ++j, ++k ) scratch[k] = values[j]; // high interval

  // Copy from scratch back to values
  for ( i=low; i<=high; ++i ) values[i] = scratch[i];

}
```

14.6 Example: Nonlinear Word Search

- What did we need to think about to **Get the Details Right** when we finished the implementation of the nonlinear word search program? What did we worry about when writing the first draft code (a.k.a. pseudo-code)? When debugging, what test cases should we be sure to try? Let's try to break the code and write down all the "corner cases" we need to test.

```
bool search_from_loc(loc position, const vector<string>& board, const string& word, vector<loc>& path) {

  // start by adding this location to the path
  path.push_back(position);
  // BASE CASE: if the path length matches the word length, we're done!
  if (path.size() == word.size()) return true;

  //  search all the places you can get to in one step
  for (int i = position.row-1; i <= position.row+1; i++) {
    for (int j = position.col-1; j <= position.col+1; j++) {
      // don't walk off the board though!
      if (i < 0 || i >= board.size()) continue;
      if (j < 0 || j >= board[0].size()) continue;
      // don't consider locations already on our path
      if (on_path(loc(i,j),path)) continue;
      // if this letter matches, recurse!
      if (word[path.size()] == board[i][j]) {
        // if we find the remaining substring, we're done!
        if (search_from_loc (loc(i,j),board,word,path))
          return true;
      }
    }
  }

  // We have failed to find a path from this loc, remove it from the path
  path.pop_back();
  return false;
}
```

14.7 Exercise: Maximum Subsequence Sum

- Problem: Given is a sequence of n values, $a_0, \ldots, a_{n-1}$, find the maximum value of $\sum_{i=j}^{k} a_i$ over all possible subsequences $j \ldots k$.
- For example, given the integers: $14, -4, 6, -9, -8, 8, -3, 16, -4, 12, -7, 4$
  The maximum subsequence sum is: $8 + (-3) + 16 + (-4) + 12 = 29$.
- Let's write a first draft of the code, and then talk about how to make it more efficient.

```
int main() {
  std::vector<int> v;
  int x;
  while (std::cin >> x) {
    v.push_back(x);
  }
```

14.8 Problem Solving Strategies

Here is an outline of the major steps to use in solving programming problems:

1. Before getting started: study the requirements, carefully!
2. Get started:
   (a) What major operations are needed and how do they relate to each other as the program flows?
   (b) What important data / information must be represented? How should it be represented? Consider and analyze several alternatives, thinking about the most important operations as you do so.
   (c) Develop a rough sketch of the solution, and write it down. There are advantages to working on paper first. Don't start hacking right away!
3. Review: reread the requirements and examine your design. Are there major pitfalls in your design? Does everything make sense? Revise as needed.
4. Details, level 1:
   (a) What major classes are needed to represent the data / information? What standard library classes can be used entirely or in part? Evaluate these based on efficiency, flexibility and ease of programming.
   (b) Draft the main program, defining variables and writing function prototypes as needed.
   (c) Draft the class interfaces — the member function prototypes.

   These last two steps can be interchanged, depending on whether you feel the classes or the main program flow is the more crucial consideration.
5. Review: reread the requirements and examine your design. Does everything make sense? Revise as needed.
6. Details, level 2:
   (a) Write the details of the classes, including member functions.
   (b) Write the functions called by the main program. Revise the main program as needed.
7. Review: reread the requirements and examine your design. Does everything make sense? Revise as needed.
8. Testing:
   (a) Test your classes and member functions. Do this separately from the rest of your program, if practical. Try to test member functions as you write them.
   (b) Test your major program functions. Write separate "driver programs" for the functions if possible. Use the debugger and well-placed output statements and output functions (to print entire classes or data structures, for example).
   (c) Be sure to test on small examples and boundary conditions.

   The goal of testing is to incrementally figure out what works — line-by-line, class-by-class, function-by-function. When you have incrementally tested everything (and fixed mistakes), the program will work.

Notes

- For larger programs and programs requiring sophisticated classes / functions, these steps may need to be repeated several times over.
- Depending on the problem, some of these steps may be more important than others.
  - For some problems, the data / information representation may be complicated and require you to write several different classes. Once the construction of these classes is working properly, accessing information in the classes may be (relatively) trivial.
  - For other problems, the data / information representation may be straightforward, but what's computed using them may be fairly complicated.
  - Many problems require combinations of both.

14.9 Design Example: Conway's Game of Life

Let's design a program to simulate Conway's Game of Life. Initially, due to time constraints, we will focus on the main data structures of needed to solve the problem.

Here is an overview of the Game:

- We have an infinite two-dimensional grid of cells, which can grow arbitrarily large in any direction.
- We will simulate the life & death of cells on the grid through a sequence of generations.
- In each generation, each cell is either alive or dead.
- At the start of a generation, a cell that was dead in the previous generation becomes alive if it had exactly 3 live cells among its 8 possible neighbors in the previous generation.
- At the start of a generation, a cell that was alive in the previous generation remains alive if and only if it had either 2 or 3 live cells among its 8 possible neighbors in the previous generation.
  - With fewer than 2 neighbors, it dies of "loneliness".
  - With more than 3 neighbors, it dies of "overcrowding".
- Important note: all births & deaths occur simultaneously in all cells at the start of a generation.
- Other birth / death rules are possible, but these have proven to be a very interesting balance.
- Many online resources are available with simulation applets, patterns, and history. For example:
  http://www.math.com/students/wonders/life/life.html
  http://www.radicaleye.com/lifepage/patterns/contents.html
  http://www.bitstorm.org/gameoflife/
  http://en.wikipedia.org/wiki/Conway's_Game_of_Life

Applying the Problem Solving Strategies

In class we will brainstorm about how to write a simulation of the Game of Life, focusing on the representation of the grid and on the actual birth and death processes.

Understanding the Requirements

We have already been working toward understanding the requirements. This effort includes playing with small examples by hand to understand the nature of the game, and a preliminary outline of the major issues.

Getting Started

- What are the important operations?
- How do we organize the operations to form the flow of control for the main program?
- What data/information do we need to represent?
- What will be the main challenges for this implementation?

Details

- New Classes? Which STL classes will be useful?

Testing

- Test Cases?

CSCI-1200 Data Structures — Spring 2017
Lecture 15 — Problem Solving Techniques, Continued

Review of Lecture 14

- General Problem Solving Techniques:
  1. Generating and Evaluating Ideas
  2. Mapping Ideas into Code
  3. Getting the Details Right
- Small exercises to practice these techniques
- Problem Solving Strategies / Checksheet

Today!

- More on Complexity
- Problem Solving Example: Quicksort (& compare to Mergesort)
- Design Example: Conway's Game of Life

Clearing Up Exponential Complexity

- Last time the instructors got tripped up, so let's start by quickly fixing our understanding of $O(s^8)$ vs $O(8^s)$.
- Recall that in the non-linear word search, from any position there are a maximum of 8 choices, so any recursive call can lead to up to 8 more!
- Remember the board is w wide, h high, and we are searching for a word with length s.
- For s=1 and an initial position, there's no recursion. Either we found the correct letter, or we didn't.
- For s=2, and an initial position (i, j), there are 8 calls: $(i-1, j-1)(i-1, j)(i-1, j+1)(i, j+1), (i+1, j+1), (i+1, j)(i+1, j-1), (i, j-1)$. This is $8^1 = 8$ calls.
- Now consider s=3. For each of the 8 positions from s=2, we can try 8 more positions. So that's $8 \times 8 = 8^2 = 64$ total calls.
- For s=i, we could repeat this, each time we're multiplying by another 8, because every position from s=$i-1$ can try 8 more positions.
- In general, our solution looks like $8^{(s-1)} = 8^s * 8^-1$. Since 8^{-1} is just a constant, we can say $O(8^s)$.
- This isn't the whole picture though. Let's consider a few cases:
  - $w \times h = 50,000$, $s = 2$? $s = 4$? $s = 50,000$?
  - $w \times h = 4$, $s = 2$? $s = 4$? $s = 50,000$?
- How we would write a recursion to be $O(s^8)$?

```
int func(int s, int layer){
if(layer==0){ return 1; }

int ret = 0;
//Make s calls
for(int i=0; i<s; i++){
ret += func(s,layer-1);
}
return ret;
}

func(1,8); => 1
func(2,8); => 256
func(3,8); => 6561
func(4,8); => 65536
```

15.1 Example: Quicksort

- Quicksort also the partition-exchange sort is another efficient sorting algorithm. Like mergesort, it is a divide and conquer algorithm.
- The steps are:
  1. Pick an element, called a pivot, from the array.
  2. Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
  3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

```
// Choose a "pivot" and rearrange the vector.  Returns the location of the
// pivot, separating top & bottom (hopefully it's near the halfway point).
int partition(vector<double>& data, int start, int end, int& swaps) {
  int mid = (start + end)/2;
  double pivot = data[mid];















  }
}

void quickSort(vector<double>& data, int start, int end) {
  if(start < end) {
    int pIndex = partition(data, start, end);
    // after calling partition, one element (the "pivot") will be at its final position
    quickSort(data, start, pIndex-1);
    quickSort(data, pIndex+1, end);
  }
}

void quickSort(vector<double>& data) {
  quickSort(data,0,data.size()-1);
}
```

- What value should you choose as the pivot? What are our different options?

- What is the order notation for the running time of this algorithm?
  What is the order notation for the additional memory use of this algorithm?

- What is the best case for this algorithm? What is the worst case for this algorithm?

- Compare the design of Quicksort and Mergesort. What is the same? What is different?

15.2 Design Example: Conway's Game of Life

Let's design a program to simulate Conway's Game of Life. Initially, due to time constraints, we will focus on the main data structures of needed to solve the problem.

Here is an overview of the Game:

- We have an infinite two-dimensional grid of cells, which can grow arbitrarily large in any direction.
- We will simulate the life & death of cells on the grid through a sequence of generations.
- In each generation, each cell is either alive or dead.
- At the start of a generation, a cell that was dead in the previous generation becomes alive if it had exactly 3 live cells among its 8 possible neighbors in the previous generation.
- At the start of a generation, a cell that was alive in the previous generation remains alive if and only if it had either 2 or 3 live cells among its 8 possible neighbors in the previous generation.
  - With fewer than 2 neighbors, it dies of "loneliness".
  - With more than 3 neighbors, it dies of "overcrowding".
- Important note: all births & deaths occur simultaneously in all cells at the start of a generation.
- Other birth / death rules are possible, but these have proven to be a very interesting balance.
- Many online resources are available with simulation applets, patterns, and history. For example:
  `http://www.math.com/students/wonders/life/life.html`
  `http://www.radicaleye.com/lifepage/patterns/contents.html`
  `http://www.bitstorm.org/gameoflife/`
  `http://en.wikipedia.org/wiki/Conway's_Game_of_Life`

Applying the Problem Solving Strategies

In class we will brainstorm about how to write a simulation of the Game of Life, focusing on the representation of the grid and on the actual birth and death processes.

Understanding the Requirements

We have already been working toward understanding the requirements. This effort includes playing with small examples by hand to understand the nature of the game, and a preliminary outline of the major issues.

Getting Started

- What are the important operations?
- How do we organize the operations to form the flow of control for the main program?
- What data/information do we need to represent?
- What will be the main challenges for this implementation?

Details

- New Classes? Which STL classes will be useful?

Testing

- Test Cases?

15.3 Generating Ideas

- If running time & memory are not primary concerns, and the problems are small, what is the simplest strategy to make sure all solutions are found. Can you write a *simple* program that tries *all possibilities*?
- What variables will control the running time & memory use of this program? What is the order notation in terms of these variables for running time & memory use?
- What incremental (baby step) improvements can be made to the naive program? How will the order notation be improved?

15.4 Mapping Ideas to Code

- What are the key steps to solving this problem? How can these steps be organized into functions and flow of control for the main function?
- What information do we need to store? What C++ or STL data types might be helpful? What new classes might we want to implement?

15.5 Getting the Details Right

- What are the simplest test cases we can start with (to make sure the control flow is correct)?

- What are some specific (simple) corner test cases we should write so we won't be surprised when we move to bigger test cases?

- What are the limitations of our approach? Are there certain test cases we won't handle correctly?

- What is the maximum test case that can be handled in a reasonable amount of time? How can we measure the performance of our algorithm & implementation?

CSCI-1200 Data Structures — Spring 2017
Lecture 16 – Associative Containers (Maps), Part 1

Review from Lectures 14 & 15

- How to design and implement algorithms using three steps or stages:
  1. Generating and Evaluating Ideas
  2. Mapping Ideas into Code
  3. Getting the Details Right
- Lots of Examples

Today's Class — Associative Containers (STL Maps)

- STL Maps: associative containers for fast insert, access and remove
- Example: Counting word occurrences
- STL Pairs
- Map iterators
- Map member functions: `operator[]`, `find`, `insert`, `erase`.
- Efficiency
- STL maps vs. STL vectors vs. STL lists

16.1 STL Maps: Associative Containers

- STL maps store pairs of "associated" values.
- We will see several examples today, in lab 9, and in Lecture 17:
  - An association between a string, representing a word, and an int representing the number of times that word has been seen in an input file.
  - An association between a string, representing a word, and a vector that stores the line numbers from a text file on which that string occurs (next lecture).
  - An association between a phone number and the name of the person with that number (tomorrow's lab).
  - An association between a class object representing a student name and the student's info (next lecture).
- A particular instance of a `map` is defined (declared) with the syntax:

  ```
  std::map<key_type, value_type>  var_name
  ```

  In our first two examples above, `key_type` is a string. In the first example, the `value_type` is an `int` and in the second it is a `std::vector<int>`.
- Entries in maps are *pairs*:

  ```
  std::pair<const key_type, value_type>
  ```

- Map iterators refer to pairs.
- Map search, insert and erase are all very fast: $O(\log n)$ time, where n is the number of pairs stored in the map.
- Note: The STL `map` type has similarities to the Python dictionary, Java HashMap, or a Perl hash, but the data structures *are not the same*. The organization, implementation, and performance is different. In a couple weeks we'll see an STL data structure that is even more similar to the Python dictionary.
- Map search, insert and erase are $O(\log n)$. Python dictionaries are $O(1)$.

First, let's see how this some of this works with a program to count the occurrences of each word in a file. We'll look at more details and more examples later.

16.2 Counting Word Occurrences

- Here's a simple and elegant solution to this problem using a map:

```
#include <iostream>
#include <map>
#include <string>

int main() {
  std::string s;
  std::map<std::string, int> counters; // store each word and an associated counter

  // read the input, keeping track of each word and how often we see it
  while (std::cin >> s)
    ++counters[s];

  // write the words and associated counts
  std::map<std::string, int>::const_iterator it;
  for (it = counters.begin(); it != counters.end(); ++it) {
    std::cout << it->first << "\t" << it->second << std::endl;
  }
  return 0;
}
```

16.3 Maps: Uniqueness and Ordering

- Maps are ordered by increasing value of the `key`. Therefore, there must be an `operator<` defined for the key.
- Once a key and its value are entered in the map, the key can't be changed. It can only be erased (together with the associated value).
- Duplicate keys can not be in the map.

map<string, int> counters

first	second
"run"	1
it → "see"	2
"spot"	1

16.4 STL Pairs

The mechanics of using `std::pair`s are relatively straightforward:

- `std::pair`s are a templated `struct` with just two members, called `first` and `second`. *Reminder: a* `struct` *is basically a wimpy* `class` *and in this course you aren't allowed to create new structs. You should use classes instead.*
- To work with pairs, you must `#include <utility>`. Note that the header file for maps (`#include <map>`) itself includes utility, so you don't have to include utility explicitly when you use pairs with maps.
- Here are simple examples of manipulating pairs:

```
std::pair<int, double> p1(5, 7.5);
std::pair<int, double> p2 = std::make_pair(8, 9.5);
p1.first = p2.first;
p2.second = 13.3;
std::cout << p1.first << " " << p1.second << std::endl;
std::cout << p2.first << " " << p2.second << std::endl;
p1 = p2;

std::pair<const std::string, double> p3 = std::make_pair(std::string("hello"), 3.5);
p3.second = -1.5;
// p3.first = std::string("illegal");      // (a)
// p1 = p3;                                // (b)
```

- The function `std::make_pair` creates a pair object from the given values. It is really just a simplified constructor, and as the example shows there are other ways of constructing pairs.
- Most of the statements in the above code show accessing and changing values in pairs.

- The two statements at the end are commented out because they cause syntax errors:
  - In (a), the `first` entry of `p3` is `const`, which means it can't be changed.
  - In (b), the two pairs are different types! Make sure you understand this.
- Returning to maps, each entry in the map is a pair object of type:

```
std::pair<const key_type, value_type>
```

  The const is needed to ensure that the keys aren't changed! This is crucial because maps are sorted by keys!

16.5 Maps: `operator[]`

- We've used the `[]` operator on vectors, which is conceptually very simple because vectors are just resizable arrays. Arrays and vectors are efficient *random access data structures*.
- But `operator[]` is actually a function call, so it can do things that aren't so simple too, for example:

```
++counters[s];
```

- For maps, the `[]` operator searches the map for the `pair` containing the `key` (string) `s`.
  - If such a pair containing the key is **not** there, the operator:
    1. creates a `pair` containing the key and a default initialized value,
    2. inserts the `pair` into the map in the appropriate position, and
    3. returns a reference to the value stored in this new pair (the second component of the pair).

    This second component may then be changed using operator++.
  - If a pair containing the key **is** there, the operator simply returns a reference to the value in that pair.
- In this particular example, the result in either case is that the `++` operator increments the value associated with string `s` (to 1 if the string wasn't already it a pair in the map).
- For the user of the map, `operator[]` makes the map feel like a vector, except that indexing is based on a `string` (or any other key) instead of an int.
- Note that the result of using [] is that the key is ALWAYS in the map afterwards.

16.6 Map Iterators

- Iterators may be used to access the map contents sequentially. Maps provide `begin()` and `end()` functions for accessing the bounding iterators. Map iterators have `++` and `--` operators.
- Each iterator refers to a pair stored in the map. Thus, given map iterator `it`, `it->first` is a `const string` and `it->second` is an `int`. Notice the use of `it->` , and remember it is just shorthand for `(*it).`

16.7 Exercise

Write code to create a map where the key is an integer and the value is a double. (Yes, an integer key!) Store each of the following in the map: 100 and its sqrt, 100,000 and its sqrt, 5 and its sqrt, and 505 and its sqrt. Write code to output the contents of the map. Draw a picture of the map contents. What will the output be?

16.8 Map Find

- One of the problems with `operator[]` is that it always places a key / value pair in the map. Sometimes we don't want this and instead we just want to check if a key is there.
- The `find` member function of the map class does this for us. For example:

```
m.find(key);
```

  where `m` is the map object and `key` is the search key. It returns a map iterator:

  If the key is in one of the pairs stored in the map, `find` returns an iterator referring to this pair.
  If the key is not in one of the pairs stored in the map, `find` returns `m.end()`.

16.9 Map Insert

- The prototype for the map `insert` member function is:

```
m.insert(std::make_pair(key, value));
```

  insert returns a pair, but not the pair we might expect. Instead it is pair of a map iterator and a bool:

```
std::pair<map<key_type, value_type>::iterator, bool>
```

- The `insert` function checks to see if the key being inserted is already in the map.
  - If so, it does not change the value, and returns a (new) pair containing an iterator referring to the *existing pair* in the map and the bool value `false`.
  - If not, it enters the pair in the map, and returns a (new) pair containing an iterator referring to the *newly added pair* in the map and the bool value `true`.

16.10 Map Erase

Maps provide three different versions of the erase member function:

- `void erase(iterator p)` — erase the pair referred to by iterator `p`.
- `void erase(iterator first, iterator last)` — erase all pairs from the map starting at `first` and going up to, but not including, `last`.
- `size_type erase(const key_type& k)` — erase the pair containing key `k`, returning either 0 or 1, depending on whether or not the key was in a pair in the map

16.11 Exercise

Re-write the `word_count` program so that it uses `find` and `insert` instead of `operator[]`.

16.12 Choices of Containers

- We can solve this word counting problem using several different approaches and different containers:
  - a vector or list of strings
  - a vector or list of pairs (string and int)
  - a map
  - ?
- How do these approaches compare? Which is cleanest, easiest, and most efficient, etc.?

CSCI-1200 Data Structures — Spring 2017
Lecture 17 – Associative Containers (Maps), Part 2

Review of Lecture 16

- Maps are associations between keys and values.
- Maps have fast insert, access and remove operations: O(log n), we'll learn why next week when we study the implementation!
- Maps store pairs; map iterators refer to these pairs.
- The primary map member functions we discussed are `operator[]`, `find`, `insert`, and `erase`.
- The choice between maps, vectors and lists is based on naturalness, ease of programming, and efficiency of the resulting program.

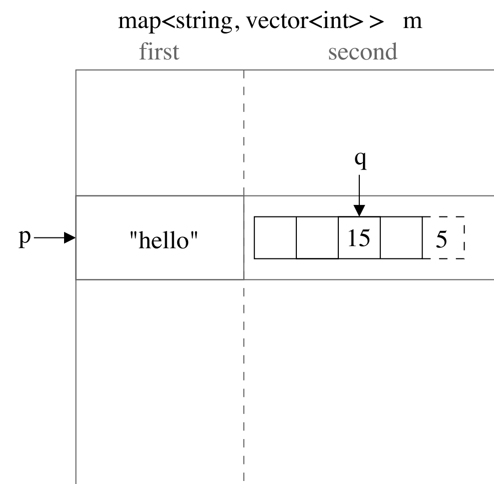
16.12 Choices of Containers

- We can solve this word counting problem using several different approaches and different containers:
  - a vector or list of strings
  - a vector or list of pairs (string and int)
  - a map
  - ?
- How do these approaches compare? Which is cleanest, easiest, and most efficient, etc.?

Today's Class — Maps, Part 2

- Maps containing more complicated values.
- Example: index mapping words to the text line numbers on which they appear.
- Maps whose keys are class objects, example: maintaining student records.
- Lists vs. Graphs vs. Trees
- Intro to Binary Trees, Binary Search Trees, & Balanced Trees

17.1 More Complicated Values



- Let's look at the example:

```
map<string, vector<int> > m;
map<string, vector<int> >::iterator p;
```

Note that the space between the `> >` is **required** (by many compiler parsers).
Otherwise, `>>` is treated as an operator.

- Here's the syntax for entering the number 5
  in the vector associated with the string "hello":

  ```
  m[string("hello")].push_back(5);
  ```

- Here's the syntax for accessing the size of the vector stored
  in the map pair referred to by map iterator p:

  ```
  p = m.find(string("hello"));
  p->second.size()
  ```

  Now, if you want to access (and change) the $\mathtt{i}^{th}$ entry in this vector you can either using subscripting:

  ```
  (p->second)[i] = 15;
  ```

  (the parentheses are needed because of precedence) or you can use vector iterators:

  ```
  vector<int>::iterator q = p->second.begin() + i;
  *q = 15;
  ```

  Both of these, of course, assume that at least `i+1` integers have been stored in the vector (either through the use of `push_back` or through construction of the vector).

- We can figure out the correct syntax for all of these by drawing pictures to visualize the contents of the map and the pairs stored in the map. We will do this during lecture, and you should do so **all the time** in practice.

17.2 Exercise

Write code to count the odd numbers stored in the map

```
  map<string, vector<int> > m;
```

This will require testing all contents of each vector in the map. Try writing the code using subscripting on the vectors and then again using vector iterators.

17.3 A Word Index in a Text File

```
// Given a text file, generate an alphabetical listing of the words in the file
// and the file line numbers on which each word appears.  If a word appears on
// a line more than once, the line number is listed only once.
#include <algorithm>
#include <cctype>
#include <iostream>
#include <map>
#include <string>
#include <vector>
using namespace std;


// implementation omitted, will be covered in a later lecture
vector<string> breakup_line_into_strings(const string& line);

int main() {
  map<string, vector<int> > words_to_lines;
  string line;
  int line_number = 0;

  while (getline(cin, line)) {
    line_number++;
    // Break the string up into words
    vector<string> words = breakup_line_into_strings(line);
```

```
    // Find if each word is already in the map.
    for (vector<string>::iterator p = words.begin(); p!= words.end(); ++p) {
      // If not, create a new entry with an empty vector (default) and
      // add to index to the end of the vector
      map<string, vector<int> >::iterator map_itr = words_to_lines.find(*p);
      if (map_itr == words_to_lines.end())
        words_to_lines[*p].push_back(line_number);  // could use insert here
      // If it is, check the last entry to see if the line number is
      // already there.  If not, add it to the back of the vector.
      else if (map_itr->second.back() != line_number)
        map_itr->second.push_back(line_number);
    }
  }

  // Output each word on a single line, followed by the line numbers.
  map<string, vector<int> >::iterator map_itr;
  for (map_itr = words_to_lines.begin(); map_itr != words_to_lines.end(); map_itr++) {
    cout << map_itr->first << ":\t";
    for (unsigned int i = 0; i < map_itr->second.size(); ++i)
      cout << (map_itr->second)[ i ] << " ";
    cout << "\n";
  }
  return 0;
}
```

17.4 Our Own Class as the Map Key

- So far we have used `string` (mostly) and `int` (once) as the key in building a `map`. Intuitively, it would seem that `string` is used quite commonly.
- More generally, we can use any class we want as long as it has an `operator<` defined on it.
- Suppose we want to maintain data for students including name, address, courses, grades, and tuition fees and calculate things like GPAs, credits, and remaining required courses. We could do this by making a single Student class object that stores everything for a particular student and put that in a vector or list. Alternately, we could break the information into separate classes and use a map. First, let's look at a sketch of a few classes that can work together to store the data:

```
class Name {
public:
  Name(const string& first, const string& last) :
    m_first(first), m_last(last) {}
  const string& first() const { return m_first; }
  const string& last() const { return m_last; }

private:
  string m_first;
  string m_last;
};


class CourseGrade {
public:
  Course(const string &c_name, const string & grade) : course_name(c_name), final_grade(grade) {}
  const string & get_course_name() const { return course_name; }
  const string & get_final_grade() const { return final_grade; }

private:
  string course_name;
  string final_grade;
};
```

```
class StudentRecord {
public:
  const string& getAddress() const { return address; }
  const string& getGradeInCourse(const string &course_name) const; /* implementation omitted */
  bool hasCompletedCourse(const string &course_name) const; /* implementation omitted */
  float getGPA() const { return GPA; }
  /* additional member functions omitted */

private:
  string address;
  vector<CourseGrade> completed_coursework;
  float GPA;
  /* etc. */
};
```

- Now if we want to create a map of student names and associated student records, we need to add an `operator<` for `Name` objects. This is simple:

```
bool operator< (const Name& left, const Name& right) {
  return left.last() < right.last() ||
    (left.last() == right.last() && left.first() < right.first());
}
```

- Now we can define a map:

```
map<Name, StudentRecord> students;
```

17.5 Exercises

- First let's draw a picture of this map data structure populated with interesting data:

- So what are the advantages of organizing this data using a map in this way? Let's assume there are s students, c different classes offered at the school, each student takes up to k classes before graduation, and at most p students take a particular course.
  - Write a fragment of code to access student X's grade in course Y. What is the order notation of this operation?
  - Write a fragment of code to make a list of *all* students who have taken course Y. What is the order notation of this operation?

17.6 Typedefs

- One of the painful aspects of using maps is the syntax. For example, consider a constant iterator in a map associating strings and vectors of ints:

```
map < string, vector<int> > :: const_iterator p;
```

- Typedefs are a syntactic means of shortening this. For example, if you place the line:

```
typedef map < string, vector<int> > map_vect;
```

  before your main function (and any function prototypes), then anywhere you want the map you can just use the identifier `map_vect`:

```
map_vect :: const_iterator p;
```

  The compiler makes the substitution for you.

17.7 When to Use Maps, Reprise

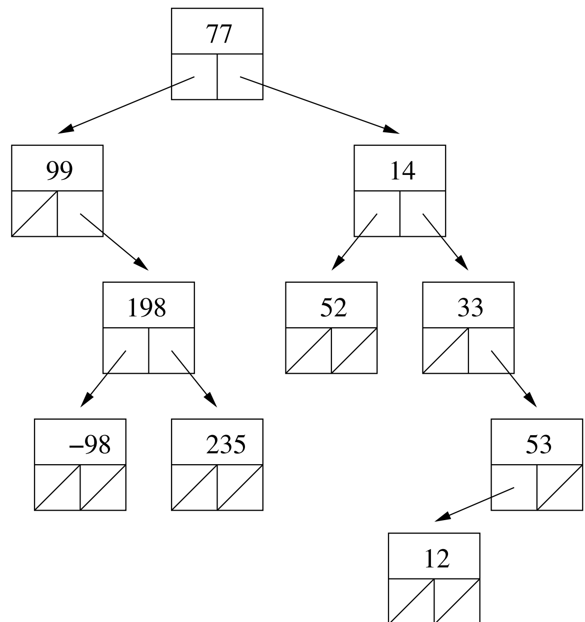
- Maps are an association between two types, one of which (the key) must have a `operator<` ordering on it.
- The association may be immediate:
  - Words and their counts.
  - Words and the lines on which they appear
- Or, the association may be created by splitting a type:
  - Splitting off the name (or student id) from rest of student record.

17.8 Overview: Lists vs. Trees vs. Graphs

- Trees create a hierarchical organization of data, rather than the linear organization in linked lists (and arrays and vectors).
- Binary search trees are the mechanism underlying maps & sets (and multimaps & multisets).
- Mathematically speaking: A *graph* is a set of vertices connected by edges. And a tree is a special graph that has no *cycles*. The edges that connect nodes in trees and graphs may be *directed* or *undirected*.

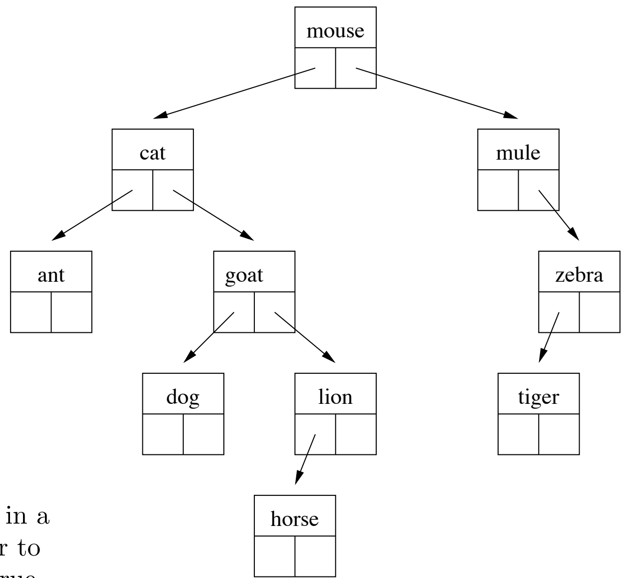
17.9 Definition: Binary Trees

- A binary tree (strictly speaking, a "rooted binary tree") is either empty or is a node that has pointers to two binary trees.
- Here's a picture of a binary tree storing integer values. In this figure, each large box indicates a tree node, with the top rectangle representing the value stored and the two lower boxes representing pointers. Pointers that are null are shown with a slash through the box.
- The topmost node in the tree is called the *root*.
- The pointers from each node are called *left* and *right*. The nodes they point to are referred to as that node's (left and right) *children*.
- The (sub)trees pointed to by the left and right pointers at *any* node are called the *left subtree* and *right subtree* of that node.
- A node where `both` children pointers are null is called a *leaf node*.
- A node's *parent* is the unique node that points to it. Only the root has no parent.



17.10 Definition: Binary Search Trees

- A *binary search tree* is a binary tree where **at each node** of the tree, the *value* stored at the node is
  - greater than or equal to all values stored in the left subtree, and
  - less than or equal to all values stored in the right subtree.
- Here is a picture of a binary search tree storing string values.



17.11 Definition: Balanced Trees

- The number of nodes on each subtree of each node in a "balanced" tree is *approximately* the same. In order to be an *exactly* balanced binary tree, what must be true about the number of nodes in the tree?
- In order to claim the performance advantages of trees, we must assume and ensure that our data structure remains approximately balanced. (You'll see much more of this in Intro to Algorithms!)

17.12 Exercise

Consider the following values:

```
4.5, 9.8, 3.5, 13.6, 19.2, 7.4, 11.7
```

1. Draw a binary tree with these values that *is NOT* a binary search tree.

2. Draw *two different* binary search trees with these values. Important note: This shows that the binary search tree structure for a given set of values is not unique!

3. How many *exactly balanced* **binary search trees** exist with these numbers? How many *exactly balanced* **binary trees** exist with these numbers?

CSCI-1200 Data Structures — Spring 2017
Lecture 18 – Trees, Part I

Review from Lectures 17

- Maps containing more complicated values. Example: index mapping words to the text line numbers on which they appear.
- Maps whose keys are class objects. Example: maintaining student records.
- Summary discussion of when to use maps.
- Lists vs. Graphs vs. Trees
- Intro to Binary Trees, Binary Search Trees, & Balanced Trees

Today's Lecture

- Finish Intro to Binary Trees, Binary Search Trees, & Balanced Trees
- STL `set` container class (like STL `map`, but without the pairs!)
- Implementation of `ds_set` class using binary search trees
- In-order, pre-order, and post-order traversal
- Breadth-first and depth-first tree search

18.1 Standard Library Sets

- STL sets are *ordered* containers storing unique "keys". An ordering relation on the keys, which defaults to `operator<`, is necessary. Because STL sets are ordered, they are technically not traditional mathematical sets.
- Sets are like maps except they have only keys, there are no associated values. Like maps, the keys are **constant**. This means you can't change a key while it is in the set. You must remove it, change it, and then reinsert it.
- Access to items in sets is extremely fast! $O(\log n)$, just like maps.
- Like other containers, sets have the usual constructors as well as the `size` member function.

18.2 Set iterators

- Set iterators, similar to map iterators, are bidirectional: they allow you to step forward (`++`) and backward (`--`) through the set. Sets provide `begin()` and `end()` iterators to delimit the bounds of the set.
- Set iterators refer to const keys (as opposed to the pairs referred to by map iterators). For example, the following code outputs all strings in the set `words`:

```
for (set<string>::iterator p = words.begin(); p!= words.end(); ++p)
  cout << *p << endl;
```

18.3 Set `insert`

- There are two different versions of the `insert` member function. The first version inserts the entry into the set and returns a pair. The first component of the returned pair refers to the location in the set containing the entry. The second component is true if the entry wasn't already in the set and therefore was inserted. It is false otherwise. The second version also inserts the key if it is not already there. The iterator `pos` is a "hint" as to where to put it. This makes the insert faster if the hint is good.

```
pair<iterator,bool> set<Key>::insert(const Key& entry);
iterator set<Key>::insert(iterator pos, const Key& entry);
```

18.4 Set `erase`

- There are three versions of `erase`. The first `erase` returns the number of entries removed (either 0 or 1). The second and third erase functions are just like the corresponding erase functions for maps. Note that the `erase` functions do not return iterators. This is different from the `vector` and `list` erase functions.

```
size_type set<Key>::erase(const Key& x);
void set<Key>::erase(iterator p);
void set<Key>::erase(iterator first, iterator last);
```

18.5 Set `find`

- The find function returns the `end` iterator if the key is not in the set:

```
const_iterator set<Key>::find(const Key& x) const;
```

18.6 Beginning our implementation of `ds_set`: The Tree Node Class

- Here is the class definition for nodes in the tree. We will use this for the tree manipulation code we write.

```
template <class T> class TreeNode {
public:
  TreeNode() : left(NULL), right(NULL) {}
  TreeNode(const T& init) : value(init), left(NULL), right(NULL) {}
  T value;
  TreeNode* left;
  TreeNode* right;
};
```

- Note: Sometimes a 3rd pointer — to the parent TreeNode — is added.

18.7 Exercises

1. Write a templated function to find the smallest value stored in a binary search tree whose root node is pointed to by `p`.

2. Write a function to count the number of odd numbers stored in a binary tree (not necessarily a binary search tree) of integers. The function should accept a `TreeNode<int>` pointer as its sole argument and return an integer. Hint: think recursively!

18.8 `ds_set` and Binary Search Tree Implementation

- A partial implementation of a set using a binary search tree is in the code attached. We will continue to study this implementation in tomorrow's lab & the next lecture.
- The increment and decrement operations for iterators have been omitted from this implementation. Next week in lecture we will discuss a couple strategies for adding these operations.
- We will use this as the basis both for understanding an initial selection of tree algorithms and for thinking about how standard library sets really work.

18.9 `ds_set`: Class Overview

- There is two auxiliary classes, `TreeNode` and `tree_iterator`. All three classes are templated.
- The only member variables of the `ds_set` class are the root and the size (number of tree nodes).
- The iterator class is declared internally, and is effectively a wrapper on the TreeNode pointers.
  - Note that `operator*` returns a `const` reference because the keys can't change.
  - The increment and decrement operators are missing (we'll fill this in next week in lecture!).
- The main public member functions just call a private (and often recursive) member function (passing the root node) that does all of the work.
- Because the class stores and manages dynamically allocated memory, a copy constructor, `operator=`, and destructor must be provided.

18.10 Exercises

1. Provide the implementation of the member function `ds_set<T>::begin`. This is essentially the problem of finding the node in the tree that stores the smallest value.

2. Write a recursive version of the function `find`.

18.11 In-order, Pre-Order, Post-Order Traversal

- One of the fundamental tree operations is "traversing" the nodes in the tree and doing something at each node. The "doing something", which is often just printing, is referred to generically as "visiting" the node.
- There are three general orders in which binary trees are traversed: pre-order, in-order and post-order.
- In order to explain these, let's first draw an "exactly balanced" binary search tree with the elements 1-7:
  - What is the *in-order traversal* of this tree? Hint: it is monotonically increasing, which is always true for an in-order traversal of a binary search tree!
  - What is the *post-order traversal* of this tree? Hint, it ends with "4" and the 3rd element printed is "2".

– What is the *pre-order traversal* of this tree? Hint, the last element is the same as the last element of the in-order traversal (but that is not true in general! why not?)

- Now let's write code to print out the elements in a binary tree in each of these three orders. These functions are easy to write recursively, and the code for the three functions looks amazingly similar. Here's the code for an in-order traversal to print the contents of a tree:

```
void print_in_order(ostream& ostr, const TreeNode<T>* p) {
  if (p) {
    print_in_order(ostr, p->left);
    ostr << p->value << "\n";
    print_in_order(ostr, p->right);
  }
}
```

  How would you modify this code to perform pre-order and post-order traversals?

18.12 Depth-first vs. Breadth-first Search

- We should also discuss two other important tree traversal terms related to problem solving and searching.
  - In a *depth-first* search, we greedily follow links down into the tree, and don't backtrack until we have hit a leaf.

    When we hit a leaf we step back out, but only to the last decision point and then proceed to the next leaf.

    This search method will quickly investigate leaf nodes, but if it has made "incorrect" branch decision early in the search, it will take a long time to work back to that point and go down the "right" branch.

  - In a *breadth-first* search, the nodes are visited with priority based on their distance from the root, with nodes closer to the root visited first.

    In other words, we visit the nodes by level, first the root (level 0), then all children of the root (level 1), then all nodes 2 links from the root (level 2), etc.

    If there are multiple solution nodes, this search method will find the solution node with the shortest path to the root node.

    However, the breadth-first search method is memory-intensive, because the implementation must store all nodes at the current level – and the worst case number of nodes on each level doubles as we progress down the tree!

- Both depth-first and breadth-first will eventually visit all elements in the tree.
- Note: The ordering of elements visited by depth-first and breadth-first is not fully specified.
  - In-order, pre-order, and post-order are all *examples* of depth-first tree traversals.
  - What is a breadth-first traversal of the elements in our sample binary search tree above? (We'll write and discuss code for breadth-first traversal next lecture!)

```
// Partial implementation of binary-tree based set class similar to std::set.
// The iterator increment & decrement operations have been omitted.
#ifndef ds_set_h_
#define ds_set_h_
#include <iostream>
#include <utility>

// -------------------------------------------------------------------
// TREE NODE CLASS
template <class T>
class TreeNode {
public:
  TreeNode() : left(NULL), right(NULL) {}
  TreeNode(const T& init) : value(init), left(NULL), right(NULL) {}
  T value;
  TreeNode* left;
  TreeNode* right;
};

template <class T> class ds_set;

// -------------------------------------------------------------------
// TREE NODE ITERATOR CLASS
template <class T>
class tree_iterator {
public:
  tree_iterator() : ptr_(NULL) {}
  tree_iterator(TreeNode<T>* p) : ptr_(p) {}
  tree_iterator(const tree_iterator& old) : ptr_(old.ptr_) {}
  ~tree_iterator() {}
  tree_iterator& operator=(const tree_iterator& old) { ptr_ = old.ptr_;  return *this; }
  // operator* gives constant access to the value at the pointer
  const T& operator*() const { return ptr_->value; }
  // comparions operators are straightforward
  bool operator==(const tree_iterator& r) { return ptr_ == r.ptr_; }
  bool operator!=(const tree_iterator& r) { return ptr_ != r.ptr_; }
  // increment & decrement will be discussed in Lecture 19 and Lab 11

private:
  // representation
  TreeNode<T>* ptr_;
};

// -------------------------------------------------------------------
// DS SET CLASS
template <class T>
class ds_set {
public:
  ds_set() : root_(NULL), size_(0) {}
  ds_set(const ds_set<T>& old) : size_(old.size_) {
    root_ = this->copy_tree(old.root_); }
  ~ds_set() { this->destroy_tree(root_);  root_ = NULL; }
  ds_set& operator=(const ds_set<T>& old) {
    if (&old != this) {
      this->destroy_tree(root_);
      root_ = this->copy_tree(old.root_);
      size_ = old.size_;
    }
    return *this;
  }

  typedef tree_iterator<T> iterator;

  int size() const { return size_; }
  bool operator==(const ds_set<T>& old) const { return (old.root_ == this->root_); }
```

```
  // FIND, INSERT & ERASE
  iterator find(const T& key_value) { return find(key_value, root_); }
  std::pair< iterator, bool > insert(T const& key_value) { return insert(key_value, root_); }
  int erase(T const& key_value) { return erase(key_value, root_); }

  // OUTPUT & PRINTING
  friend std::ostream& operator<< (std::ostream& ostr, const ds_set<T>& s) {
    s.print_in_order(ostr, s.root_);
    return ostr;
  }
  void print_as_sideways_tree(std::ostream& ostr) const { print_as_sideways_tree(ostr, root_, 0); }

  // ITERATORS
  iterator begin() const {
    // Implemented in Lecture 18









  }
  iterator end() const { return iterator(NULL); }

private:
  // REPRESENTATION
  TreeNode<T>* root_;
  int size_;

  // PRIVATE HELPER FUNCTIONS
  TreeNode<T>*  copy_tree(TreeNode<T>* old_root) {  /* Implemented in Lab 10 */  }
  void destroy_tree(TreeNode<T>* p) {  /* Implemented in Lecture 19 */  }

  iterator find(const T& key_value, TreeNode<T>* p) {
    // Implemented in Lecture 18










  }

  std::pair<iterator,bool> insert(const T& key_value, TreeNode<T>*& p) {  /* Discussed in Lecture 19 */  }
  int erase(T const& key_value, TreeNode<T>* &p) {  /* Implemented in Lecture 19 */  }

  void print_in_order(std::ostream& ostr, const TreeNode<T>* p) const {
    // Discussed in Lecture 18
    if (p) {
      print_in_order(ostr, p->left);
      ostr << p->value << "\n";
      print_in_order(ostr, p->right);
    }
  }

  void print_as_sideways_tree(std::ostream& ostr, const TreeNode<T>* p, int depth) const {
    /* Discussed in Lecture 19 */  }
};

#endif
```

CSCI-1200 Data Structures — Spring 2017
Lecture 19 – Trees, Part II

Review from Lecture 18 and Lab 10

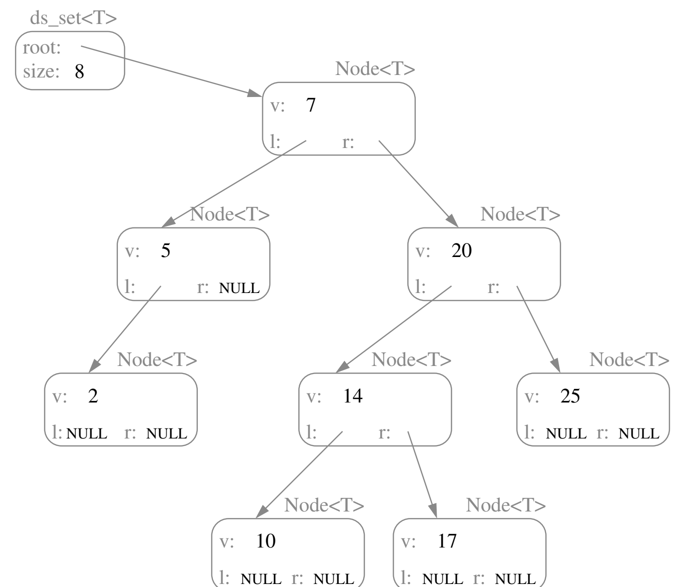
- Binary Trees, Binary Search Trees, & Balanced Trees
- STL `set` container class (like STL `map`, but without the pairs!)
- Finding the smallest element in a BST.
- Overview of the `ds_set` implementation: `begin` and `find`.

Today's Lecture

- Warmup / Review: `destroy_tree`
- A very important `ds_set` operation insert
- In-order, pre-order, and post-order traversal; Breadth-first and depth-first tree search
- Finding the *in-order successor* of a binary tree node, tree itertor increment

19.1 Warmup Exercise

- Write the `ds_set::destroy_tree` private helper function.



19.2 Insert

- Move left and right down the tree based on comparing keys. The goal is to find the location to do an insert *that preserves the binary search tree ordering property.*
- We will always be inserting at an empty (NULL) pointer location.
- **Exercise:** Why does this work? Is there always a place to put the new item? Is there ever more than one place to put the new item?

- IMPORTANT NOTE: Passing pointers by reference ensures that the new node is truly inserted into the tree. This is subtle but important.
- Note how the return value pair is constructed.
- **Exercise:** How does the order that the nodes are inserted affect the final tree structure? Give an ordering that produces a balanced tree and an insertion ordering that produces a highly unbalanced tree.

19.3 In-order, Pre-order, Post-order Traversal

- Reminder: For an exactly balanced binary search tree with the elements 1-7:
  - In-order: `1 2 3  (4)  5 6 7`
  - Pre-order: `(4)  2 1 3  6 5 7`
  - Post-order: `1 3 2  5 7 6  (4)`
- Now let's write code to print out the elements in a binary tree in each of these three orders. These functions are easy to write recursively, and the code for the three functions looks amazingly similar. Here's the code for an in-order traversal to print the contents of a tree:

```
void print_in_order(ostream& ostr, const TreeNode<T>* p) {
  if (p) {
    print_in_order(ostr, p->left);
    ostr << p->value << "\n";
    print_in_order(ostr, p->right);
  }
}
```

- How would you modify this code to perform pre-order and post-order traversals?
- What is the traversal order of the `destroy_tree` function we wrote earlier?

19.4 Depth-first vs. Breadth-first Search

- We should also discuss two other important tree traversal terms related to problem solving and searching.
  - In a *depth-first* search, we greedily follow links down into the tree, and don't backtrack until we have hit a leaf.

    When we hit a leaf we step back out, but only to the last decision point and then proceed to the next leaf.

    This search method will quickly investigate leaf nodes, but if it has made an "incorrect" branch decision early in the search, it will take a long time to work back to that point and go down the "right" branch.
  - In a *breadth-first* search, the nodes are visited with priority based on their distance from the root, with nodes closer to the root visited first.

    In other words, we visit the nodes by level, first the root (level 0), then all children of the root (level 1), then all nodes 2 links from the root (level 2), etc.

    If there are multiple solution nodes, this search method will find the solution node with the shortest path to the root node.

    However, the breadth-first search method is memory-intensive, because the implementation must store all nodes at the current level – and the worst case number of nodes on each level doubles as we progress down the tree!
- Both depth-first and breadth-first will eventually visit all elements in the tree.
- Note: The ordering of elements visited by depth-first and breadth-first is not fully specified.
  - In-order, pre-order, and post-order are all *examples* of depth-first tree traversals.

    Note: A simple recursive tree function is usually a depth-first traversal.
  - What is a breadth-first traversal of the elements in our sample binary search trees above?

19.5 General-Purpose Breadth-First Search/Tree Traversal

- Write an algorithm to print the nodes in the tree one tier at a time, that is, in a *breadth-first* manner.

- What is the best/average/worst-case running time of this algorithm? What is the best/average/worst-case memory usage of this algorithm? Give a specific example tree that illustrates each case.

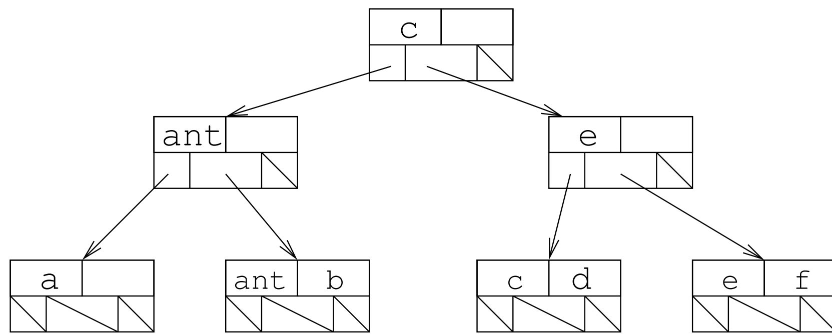
19.6 Limitations of Our BST Implementation

- The efficiency of the main insert, find and erase algorithms depends on the height of the tree.
- The best-case and average-case heights of a binary search tree storing n nodes are both $O(\log n)$. The worst-case, which often can happen in practice, is $O(n)$.
- Developing more sophisticated algorithms to avoid the worst-case behavior will be covered in Introduction to Algorithms. One elegant extension to the binary search tree is described below...

19.7 B+ Trees

- Unlike binary search trees, nodes in B+ trees (and their predecessor, the B tree) have up to b children. Thus B+ trees are very flat and very wide. This is good when it is very expensive to move from one node to another.
- B+ trees are supposed to be associative (i.e. they have key-value pairs), but we will just focus on the keys.
- Just like STL `map` and STL `set`, these *keys* and *values* can be any type, but *keys* must have an `operator<` defined.
- We can use all our normal terminology, but we'll also refer to non-leaf nodes as "internal nodes".
- In a B tree value-key pairs can show up anywhere in the tree, in a B+ tree all the key-value pairs are in the leaves and the internal nodes contain duplicates of some keys.
- In either type of tree, all leaves are the same distance from the root.
- The keys are always sorted in a B/B+ tree node, and there are up to $b-1$ of them. They act like $b-1$ binary search tree nodes mashed together.
- In fact, with the exception of the root, nodes will always have between roughly $\frac{b}{2}$ and $b-1$ keys (in our implementation).
- If a B+ tree node has k keys $key_0, key_1, key_2, \ldots, key_k$, it will have $k+1$ children. The keys in the leftmost child must be $< key_0$, the next child must have keys such that they are $\leq key_0$ and $< key_1$, and so on up to the rightmost child which has only keys $\geq key_k$.

- HW8 will focus on implementing some of the functionality of a B+ tree. It won't be enough to replace a real B+ tree, but it will be enough to understand how the tree works and construct trees.



- Considerations in a full implementation:
  - What happens when we want to add a key to a node that's already full?
  - How do we remove values from a node?
  - How do we ensure the tree stays balanced?
  - How to keep leaves linked together?
  - How to represent key-value pairs?

CSCI-1200 Data Structures — Spring 2017
Lecture 20 – Trees, Part III

Review from Lecture 18 & 19

- Overview of the `ds_set` implementation
- `begin`, `find`, `destroy_tree`, `insert`
- In-order, pre-order, and post-order traversal; Breadth-first and depth-first tree search

```
template <class T>
void breadth_first_print (TreeNode<T> *p) {
  if (p != NULL) {
    std::list<TreeNode<T>*> current_level;
    current_level.push_back(p);
    while (current_level.size() != 0) {
      std::list<TreeNode<T>*> next_level;
      for (std::list<TreeNode<T>*>::iterator itr = current_level.begin();
           itr != current_level.end(); itr++) {
        std::cout << (*itr)->value;
        if ((*itr)->left != NULL) { next_level.push_back((*itr)->left); }
        if ((*itr)->right != NULL) { next_level.push_back((*itr)->right); }
      }
      current_level = next_level;
    }
  }
}
```

- B^+ tree overview

Today's Lecture

- Iterators
- Last piece of `ds_set`: removing an item, `erase`
- Tree height, longest-shortest paths, breadth-first search
- Erase with parent pointers, increment operation on iterators
- Limitations of our `ds_set` implementation

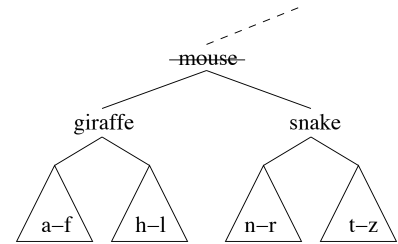
20.1 `ds_set` Warmup/Review Exercises

- Draw a diagram of a *possible* memory layout for a `ds_set` containing the numbers 16, 2, 8, 11, and 5. Is there only one valid memory layout for this data as a `ds_set`? Why?

- In what order should a forward iterator visit the data? Draw an *abstract* table representation of this data (omits details of `TreeNode` memory layout).

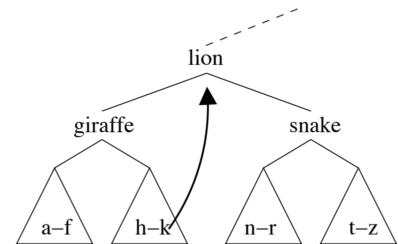
20.2 Erase

First we need to find the node to remove. Once it is found,
the actual removal is easy if the node has no children or only one child.
Draw picture of each case!



It is harder if there are two children:

- Find the node with the greatest value in the left subtree or the node with the smallest value in the right subtree.
- The value in this node may be safely moved into the current node because of the tree ordering.
- Then we recursively apply erase to remove that node — which is guaranteed to have at most one child.



Exercise: Write a recursive version of erase.

Exercise: How does the order that nodes are deleted affect the tree structure? Starting with a mostly balanced tree, give an erase ordering that yields an unbalanced tree.

20.3 Height and Height Calculation Algorithm

- The *height* of a node in a tree is the length of the longest path down the tree from that node to a leaf node. The height of a leaf is 1. We will think of the height of a null pointer as 0.
- The height of the tree is the height of the root node, and therefore if the tree is empty the height will be 0.

  Exercise: Write a simple recursive algorithm to calculate the height of a tree.

- What is the best/average/worst-case running time of this algorithm? What is the best/average/worst-case memory usage of this algorithm? Give a specific example tree that illustrates each case.

20.4 Shortest Paths to Leaf Node

- Now let's write a function to instead calculate the *shortest* path to a NULL child pointer.

- What is the running time of this algorithm? Can we do better? *Hint: How does a breadth-first vs. depth-first algorithm for this problem compare?*

20.5 Tree Iterator Increment/Decrement - Implementation Choices

- The increment operator should change the iterator's pointer to point to the next TreeNode in an in-order traversal — the "in-order successor" — while the decrement operator should change the iterator's pointer to point to the "in-order predecessor".
- Unlike the situation with lists and vectors, these predecessors and successors are not necessarily "nearby" (either in physical memory or by following a link) in the tree, as examples we draw in class will illustrate.
- There are two common solution approaches:
  - Each node stores a parent pointer. Only the root node has a null parent pointer. [method 1]
  - Each iterator maintains a stack of pointers representing the path down the tree to the current node. [method 2]
- If we choose the parent pointer method, we'll need to rewrite the `insert` and `erase` member functions to correctly adjust parent pointers.
- Although iterator increment looks expensive in the worst case for a single application of `operator++`, it is fairly easy to show that iterating through a tree storing n nodes requires $O(n)$ operations overall.

Exercise: [method 1] Write a fragment of code that given a node, finds the in-order successor using parent pointers. Be sure to draw a picture to help you understand!

Exercise: [method 2] Write a fragment of code that given a tree iterator containing a pointer to the node *and* a stack of pointers representing path from root to node, finds the in-order successor (without using parent pointers).

Either version can be extended to complete the implementation of increment/decrement for the `ds_set` *tree iterators.*

Exercise: What are the advantages & disadvantages of each method?

20.6 Erase (now with parent pointers)

- If we choose to use parent pointers, we need to add to the Node representation, and re-implement several `ds_set` member functions.
- **Exercise:** Study the new version of `insert`, with parent pointers.
- **Exercise:** Rewrite `erase`, now with parent pointers.

```
// -------------------------------------------------------------------
// TREE NODE CLASS
template <class T> class TreeNode {
public:
  TreeNode() : left(NULL), right(NULL), parent(NULL) {}
  TreeNode(const T& init) : value(init), left(NULL), right(NULL), parent(NULL) {}
  T value;
  TreeNode* left;
  TreeNode* right;
  TreeNode* parent; // to allow implementation of iterator increment & decrement
};
template <class T> class ds_set;
// -------------------------------------------------------------------
// TREE NODE ITERATOR CLASS
template <class T> class tree_iterator {
public:
  tree_iterator() : ptr_(NULL), set_(NULL) {}
  tree_iterator(TreeNode<T>* p, const ds_set<T> * s) : ptr_(p), set_(s) {}
  // operator* gives constant access to the value at the pointer
  const T& operator*() const { return ptr_->value; }
  // comparions operators are straightforward
  bool operator== (const tree_iterator& rgt) { return ptr_ == rgt.ptr_; }
  bool operator!= (const tree_iterator& rgt) { return ptr_ != rgt.ptr_; }
  // increment & decrement operators
  tree_iterator<T> & operator++() {
    if (ptr_->right != NULL) { // find the leftmost child of the right node
      ptr_ = ptr_->right;
      while (ptr_->left != NULL) { ptr_ = ptr_->left; }
    } else { // go upwards along right branches...  stop after the first left
      while (ptr_->parent != NULL && ptr_->parent->right == ptr_) { ptr_ = ptr_->parent; }
      ptr_ = ptr_->parent;
    }
    return *this;
  }
  tree_iterator<T> operator++(int) { tree_iterator<T> temp(*this);  ++(*this); return temp; }
  tree_iterator<T> & operator--() {
    if (ptr_ == NULL) { // so that it works for end()
      assert (set_ != NULL);
      ptr_ = set_->root_;
      while (ptr_->right != NULL) { ptr_ = ptr_->right; }
    } else if (ptr_->left != NULL) { // find the rightmost child of the left node
      ptr_ = ptr_->left;
      while (ptr_->right != NULL) { ptr_ = ptr_->right; }
    } else { // go upwards along left brances... stop after the first right
      while (ptr_->parent != NULL && ptr_->parent->left == ptr_) { ptr_ = ptr_->parent; }
      ptr_ = ptr_->parent;
    }
    return *this;
  }
  tree_iterator<T> operator--(int) { tree_iterator<T> temp(*this); --(*this); return temp; }
private:
  // representation
  TreeNode<T>* ptr_;
  const ds_set<T>* set_;
};
// -------------------------------------------------------------------
// DS_ SET CLASS
template <class T> class ds_set {
public:
  ds_set() : root_(NULL), size_(0) {}
  ds_set(const ds_set<T>& old) : size_(old.size_) { root_ = this->copy_tree(old.root_,NULL); }
  ~ds_set() { this->destroy_tree(root_); root_ = NULL; }
  ds_set& operator=(const ds_set<T>& old) {
    if (&old != this) {
      this->destroy_tree(root_);
```

```
      root_ = this->copy_tree(old.root_,NULL);
      size_ = old.size_;
    }
    return *this;
  }
  typedef tree_iterator<T> iterator;
  friend class tree_iterator<T>;
  int size() const { return size_; }
  bool operator==(const ds_set<T>& old) const { return (old.root_ == this->root_); }
  // FIND, INSERT & ERASE
  iterator find(const T& key_value) { return find(key_value, root_); }
  std::pair< iterator, bool > insert(T const& key_value) { return insert(key_value, root_, NULL); }
  int erase(T const& key_value) { return erase(key_value, root_); }
  // ITERATORS
  iterator begin() const {
    if (!root_) return iterator(NULL,this);
    TreeNode<T>* p = root_;
    while (p->left) p = p->left;
    return iterator(p,this);
  }
  iterator end() const { return iterator(NULL,this); }
private:
  // REPRESENTATION
  TreeNode<T>* root_;
  int size_;
  // PRIVATE HELPER FUNCTIONS
  TreeNode<T>*  copy_tree(TreeNode<T>* old_root, TreeNode<T>* the_parent) {
    if (old_root == NULL) return NULL;
    TreeNode<T> *answer = new TreeNode<T>();
    answer->value = old_root->value;
    answer->left = copy_tree(old_root->left,answer);
    answer->right = copy_tree(old_root->right,answer);
    answer->parent = the_parent;
    return answer;
  }
  void destroy_tree(TreeNode<T>* p) {
    if (!p) return;
    destroy_tree(p->right);
    destroy_tree(p->left);
    delete p;
  }
  iterator find(const T& key_value, TreeNode<T>* p) {
    if (!p) return end();
    if      (p->value > key_value) return find(key_value, p->left);
    else if (p->value < key_value) return find(key_value, p->right);
    else                           return iterator(p,this);
  }
  std::pair<iterator,bool> insert(const T& key_value, TreeNode<T>*& p, TreeNode<T>* the_parent) {
    if (!p) {
      p = new TreeNode<T>(key_value);
      p->parent = the_parent;
      this->size_++;
      return std::pair<iterator,bool>(iterator(p,this), true);
    }
    else if (key_value < p->value)
      return insert(key_value, p->left, p);
    else if (key_value > p->value)
      return insert(key_value, p->right, p);
    else
      return std::pair<iterator,bool>(iterator(p,this), false);
  }
  int erase(T const& key_value, TreeNode<T>* &p) {
    /* Implemented in Lecture 20 */
  }
};
```

CSCI-1200 Data Structures — Spring 2017
Lecture 21 – Operators & Friends

Announcements: Test 3 Information

- Test 3 will be held Monday, April 10th from 6-7:50pm.
  Your exam room & zone assignment are posted on the homework submission site.
  Note: We have re-shuffled the room & zone assignments from Exams 1 & 2.

  No make-ups will be given except for emergency situations, and even then a written excuse from the Dean of Students or the Office of Student Experience will be required.
- Coverage: Lectures 1-21, Labs 1-10, HW 1-8.
- Closed-book and closed-notes *except for 1 sheet of notes on 8.5x11 inch paper (front & back) that may be handwritten or printed.* Computers, cell-phones, calculators, music players, etc. are not permitted and must be turned off. All students must bring their Rensselaer photo ID card.
- Practice problems from previous exams are available on the course website. Solutions to the problems will be posted on Sunday evening.

Review from Lecture 20

- Last piece of `ds_set`: removing an item, `erase`
- Tree height & tree height order notation

Today's Lecture

- Finish last lecture!
  - Shortest path to leaf, iterators, representing the parent
- Operators as non-member functions, as member functions, and as friend functions.

21.1 Complex Numbers — A Brief Review

- Complex numbers take the form $z = a + bi$, where $i = \sqrt{-1}$ and a and b are real. a is called the real part, b is called the imaginary part.
- If $w = c + di$, then
  - $w + z = (a + c) + (b + d)i$,
  - $w - z = (a - c) + (b - d)i$, and
  - $w \times z = (ac - bd) + (ad + bc)i$
- The magnitude of a complex number is $\sqrt{a^2 + b^2}$.

21.2 `Complex` Class declaration (`complex.h`)

```
class Complex {
public:
  Complex(double x=0, double y=0) : real_(x), imag_(y) {}  // default constructor
  Complex(Complex const& old) : real_(old.real_), imag_(old.imag_) {}  // copy constructor
  Complex& operator= (Complex const& rhs); // Assignment operator
  double Real() const { return real_; }
  void SetReal(double x) { real_ = x; }
  double Imaginary() const { return imag_; }
  void SetImaginary(double y) { imag_ = y; }
  double Magnitude() const { return sqrt(real_*real_ + imag_*imag_); }
  Complex operator+ (Complex const& rhs) const;
  Complex operator- () const; // unary operator- negates a complex number
  friend istream& operator>> (istream& istr, Complex& c);
private:
  double real_, imag_;
};
```

```
Complex operator- (Complex const& left, Complex const& right); // non-member function
ostream& operator<< (ostream& ostr, Complex const& c);  // non-member function
```

21.3 Implementation of Complex Class (complex.cpp)

```
// Assignment operator
Complex& Complex::operator= (Complex const& rhs) {
  real_ = rhs.real_;
  imag_ = rhs.imag_;
  return *this;
}


// Addition operator as a member function.
Complex Complex::operator+ (Complex const& rhs) const {
  double re = real_ + rhs.real_;
  double im = imag_ + rhs.imag_;
  return Complex(re, im);
}
// Subtraction operator as a non-member function.
Complex operator- (Complex const& lhs, Complex const& rhs) {
  return Complex(lhs.Real()-rhs.Real(), lhs.Imaginary()-rhs.Imaginary());
}
// Unary negation operator.  Note that there are no arguments.
Complex Complex::operator- () const {
  return Complex(-real_, -imag_);
}
// Input stream operator as a friend function
istream& operator>> (istream & istr, Complex & c) {
  istr >> c.real_ >> c.imag_;
  return istr;
}
// Output stream operator as an ordinary non-member function
ostream& operator<< (ostream & ostr, Complex const& c) {
  if (c.Imaginary() < 0)  ostr << c.Real() << " - " << -c.Imaginary() << " i ";
  else                    ostr << c.Real() << " + " <<  c.Imaginary() << " i ";
  return ostr;
}
```

21.4 Operators as Non-Member Functions and as Member Functions

- We have already written our own operators, especially `operator<`, to sort objects stored in STL containers and to create our own keys for maps.
- We can write them as non-member functions (e.g., `operator-`). When implemented as a non-member function, the expression: `z - w` is translated by the compiler into the function call: `operator- (z, w)`
- We can also write them as member functions (e.g., `operator+`). When implemented as a member function, the expression: `z + w` is translated into: `z.operator+ (w)`

  This shows that `operator+` is a member function of `z`, since `z` appears on the left-hand side of the operator. Observe that the function has **only one** argument!

  There are several important properties of the implementation of an operator as a member function:

  - It is within the scope of class `Complex`, so private member variables can be accessed directly.
  - The member variables of `z`, whose member function is actually called, are referenced by directly by name.
  - The member variables of `w` are accessed through the parameter `rhs`.
  - The member function is `const`, which means that `z` will not (and can not) be changed by the function. Also, since `w` will not be changed since the argument is also marked `const`.
- Both `operator+` and `operator-` return `Complex` objects, so both must call `Complex` constructors to create these objects. Calling constructors for `Complex` objects inside functions, especially member functions that work on `Complex` objects, seems somewhat counter-intuitive at first, but it is common practice!

21.5 Assignment Operators

- The assignment operator: `z1 = z2;` becomes a function call: `z1.operator=(z2);`

  And cascaded assignments like: `z1 = z2 = z3;` are really: `z1 = (z2 = z3);`
  which becomes: `z1.operator= (z2.operator= (z3));`

  Studying these helps to explain how to write the assignment operator, which is usually a member function.

- The argument (the right side of the operator) is passed by constant reference. Its values are used to change the contents of the left side of the operator, which is the object whose member function is called. A reference to this object is returned, allowing a subsequent call to `operator=` (`z1`'s `operator=` in the example above).

  The identifier `this` is reserved as a pointer inside class scope to the object whose member function is called. Therefore, `*this` is a a reference to this object.

- The fact that `operator=` returns a reference allows us to write code of the form: `(z1 = z2).real();`

21.6 Exercise

Write an `operator+=` as a member function of the `Complex` class. To do so, you must combine what you learned about `operator=` and `operator+`. In particular, the new operator must return a reference, `*this`.

21.7 Returning Objects vs. Returning References to Objects

- In the `operator+` and `operator-` functions we create new `Complex` objects and simply return the new object. The return types of these operators are both `Complex`.

  Technically, we don't return the new object (which is stored only locally and will disappear once the scope of the function is exited). Instead we create a copy of the object and return the copy. This automatic copying happens outside of the scope of the function, so it is *safe* to access outside of the function. *Note: It's important that the copy constructor is correctly implemented!* Good compilers can minimize the amount of redundant copying without introducing semantic errors.

- When you change an existing object inside an operator and need to return that object, you must return a **reference** to that object. This is why the return types of `operator=` and `operator+=` are both `Complex&`. This avoids creation of a new object.

- A common error made by beginners (and some non-beginners!) is attempting to return a reference to a locally created object! This results in someone having a pointer to stale memory. The pointer may behave correctly for a short while... until the memory under the pointer is allocated and used by someone else.

21.8 Friend Classes vs. Friend Functions

- In the example below, the `Foo` class has designated the `Bar` to be a **friend**. This must be done in the `public` area of the declaration of `Foo`.

  ```
  class Foo {
  public:
    friend class Bar;
    ...
  };
  ```

  This allows member functions in class `Bar` to access *all of* the private member functions and variables of a `Foo` object as though they were public (but not vice versa). Note that `Foo` is giving friendship (access to its private contents) rather than `Bar` claiming it. What could go wrong if we allowed friendships to be claimed?

- Alternatively, within the definition of the class, we can designate specific functions to be "`friend`"s, which grants these functions access similar to that of a member function. The most common example of this is operators, and especially stream operators.

21.9 Stream Operators as Friend Functions

- The operators `>>` and `<<` are defined for the `Complex` class. These are binary operators.
  The compiler translates: `cout << z3` into: `operator<< (cout, z3)`
  Consecutive calls to the `<<` operator, such as: `cout << "z3 = " << z3 << endl;`
  are translated into: `((cout << "z3 = ") << z3) << endl;`
  Each application of the operator returns an `ostream` object so that the next application can occur.

- If we wanted to make one of these stream operators a regular member function, it would have to be a member function of the `ostream` class because this is the first argument (left operand). *We cannot make it a member function of the* `Complex` *class*. This is why stream operators are never member functions.
- Stream operators are either ordinary non-member functions (if the operators can do their work through the public class interface) or friend functions (if they need non public access).

21.10 Summary of Operator Overloading in C++

- Unary operators that can be overloaded: `+ - * & ~ ! ++ -- -> ->*`
- Binary operators that can be overloaded: `+ - * / % ^ & | << >> += -= *= /= %= ^= &= |= <<= >>= < <= > >= == != && || , [] () new new[] delete delete[]`
- There are only a few operators that can not be overloaded: `. .* ?: ::`
- We can't create new operators and we can't change the number of arguments (except for the function call operator, which has a variable number of arguments).
- There are three different ways to overload an operator. When there is a choice, we recommend trying to write operators in this order:
  - Non-member function
  - Member function
  - Friend function
- The most important rule for clean class design involving operators is to **NEVER change the intuitive meaning of an operator**. The whole point of operators is lost if you do. One (bad) example would be defining the increment operator on a `Complex` number.

21.11 Extra Practice

- Implement the following operators for the `Complex` class (or explain why they cannot or should not be implemented). Think about whether they should be non-member, member, or friend.

  `operator*  operator==  operator!=  operator<`

21.12 A Tree Practice Problem

- Draw a *balanced binary tree* that contains the values: 6, 13, 9, 17, 32, 23, and 20.
- What is the height of a *balanced binary tree* storing n elements?
- Draw a *binary search tree* that has *post-order traversal*: 6 13 9 17 32 23 20.
- How many other correct answers are possible for the previous question?

CSCI-1200 Data Structures — Spring 2017
Lecture 22 – Hash Tables

Review from Lecture 21

- Finishing binary search trees & the `ds_set` class
- Operators as non-member functions, as member functions.
- (Today) operators as friend functions.

Today's Lecture

- "the single most important data structure known to mankind"
- Hash Tables, Hash Functions, and Collision Resolution
- Performance of: Hash Tables vs. Binary Search Trees
- Collision resolution: separate chaining vs open addressing
- STL's `unordered_set` (and `unordered_map`)
- Using a hash table to implement a set/map
  - Hash functions as functors/function objects
  - Iterators, find, insert, and erase

22.1 Definition: What's a Hash Table?

- A table implementation with *constant time access*.
  - Like a set, we can store elements in a collection. Or like a map, we can store key-value pair associations in the hash table. But it's even faster to do find, insert, and erase with a hash table! However, hash tables *do not* store the data in sorted order.
- A hash table is implemented with an array at the top level.
- Each element or key is mapped to a slot in the array by a *hash function*.

22.2 Definition: What's a Hash Function?

- A simple function of one argument (the key) which returns an integer index (a bucket or slot in the array).
- Ideally the function will "uniformly" distribute the keys throughout the range of legal index values ($0 \rightarrow$ k-1).
- **What's a collision?**
  When the hash function maps multiple (different) keys to the same index.
- **How do we deal with collisions?**
  One way to resolve this is by storing a linked list of values at each slot in the array.

22.3 Example: Caller ID

- We are given a phonebook with 50,000 name/number pairings. Each number is a 10 digit number. We need to create a data structure to lookup the name matching a particular phone number. Ideally, name lookup should be O(1) time expected, and the caller ID system should use O(n) memory (n = 50,000).
- Note: In the toy implementations that follow we use small datasets, but we should evaluate the system scaled up to handle the large dataset.

- The basic interface:

```
// add several names to the phonebook
add(phonebook, 1111, "fred");
add(phonebook, 2222, "sally");
add(phonebook, 3333, "george");
// test the phonebook
std::cout << identify(phonebook, 2222) << " is calling!" << std::endl;
std::cout << identify(phonebook, 4444) << " is calling!" << std::endl;
```

- We'll review how we solved this problem in Lab 9 with an STL `vector` then an STL `map`. Finally, we'll implement the system with a hash table.

22.4 Caller ID with an STL Vector

```
// create an empty phonebook
std::vector<std::string> phonebook(10000, "UNKNOWN CALLER");

void add(std::vector<std::string> &phonebook, int number, std::string name) {
  phonebook[number] = name; }

std::string identify(const std::vector<std::string> &phonebook, int number) {
  return phonebook[number]; }
```

Exercise: What's the memory usage for the vector-based Caller ID system?
What's the expected running time for find, insert, and erase?

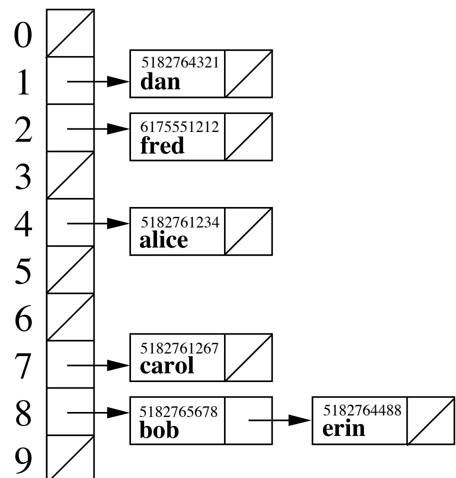
22.5 Caller ID with an STL Map

```
// create an empty phonebook
std::map<int,std::string> phonebook;

void add(std::map<int,std::string> &phonebook, int number, std::string name) {
  phonebook[number] = name; }

std::string identify(const std::map<int,std::string> &phonebook, int number) {
  map<int,std::string>::const_iterator tmp = phonebook.find(number);
  if (tmp == phonebook.end()) return "UNKNOWN CALLER"; else return tmp->second;
}
```

Exercise: What's the memory usage for the map-based Caller ID system?
What's the expected running time for find, insert, and erase?

22.6 Now let's implement Caller ID with a Hash Table

```
#define PHONEBOOK_SIZE 10

class Node {
public:
  int number;
  string name;
  Node* next;
};

// create the phonebook, initially all numbers are unassigned
Node* phonebook[PHONEBOOK_SIZE];
for (int i = 0; i < PHONEBOOK_SIZE; i++) {
  phonebook[i] = NULL;
}
```



```
// corresponds a phone number to a slot in the array
int hash_function(int number) {



}

// add a number, name pair to the phonebook
void add(Node* phonebook[PHONEBOOK_SIZE], int number, string name) {





}

// given a phone number, determine who is calling
void identify(Node* phonebook[PHONEBOOK_SIZE], int number) {







}
```

22.7 Exercise: Choosing a Hash Function

- What's a good hash function for this application?
- What's a bad hash function for this application?

22.8 Exercise: Hash Table Performance

- What's the memory usage for the hash-table-based Caller ID system?
- What's the expected running time for find, insert, and erase?

22.9 What makes a Good Hash Function?

- Goals: **fast O(1) computation** and a **random, uniform distribution of keys throughout the table**, *despite the actual distribution of keys that are to be stored.*
- For example, using: `f(k) = abs(k)%N` as our hash function satisfies the first requirement, but may not satisfy the second.

- Another example of a dangerous hash function on string keys is to add or multiply the ascii values of each char:

```
unsigned int hash(string const& k, unsigned int N) {
  unsigned int value = 0;
  for (unsigned int i=0; i<k.size(); ++i)
    value += k[i];  // conversion to int is automatic
  return k % N;
}
```

  The problem is that different permutations of the same string result in the same hash table location.

- This can be improved through multiplications that involve the position and value of the key:

```
unsigned int hash(string const& k, unsigned int N) {
  unsigned int value = 0;
  for (unsigned int i=0; i<k.size(); ++i)
    value = value*8 + k[i];  // conversion to int is automatic
  return k % N;
}
```

- The 2nd method is better, but can be improved further. The theory of good hash functions is quite involved and beyond the scope of this course.

22.10 How do we Resolve Collisions? METHOD 1: Separate Chaining

- Each table location stores a linked list of keys (and values) hashed to that location (as shown above in the phonebook hashtable). Thus, the hashing function really just selects which list to search or modify.
- This works well when the number of items stored in each list is small, e.g., an average of 1. Other data structures, such as binary search trees, may be used in place of the list, but these have even greater overhead considering the (hopefully, very small) number of items stored per bin.

22.11 How do we Resolve Collisions? METHOD 2: Open Addressing

- In *open addressing*, when the chosen table location already stores a key (or key-value pair), a different table location is sought in order to store the new value (or pair).
- Here are three different open addressing variations to handle a collision during an *insert* operation:
  - *Linear probing:* If `i` is the chosen hash location then the following sequence of table locations is tested ("probed") until an empty location is found:

    ```
    (i+1)%N, (i+2)%N, (i+3)%N, ...
    ```

  - *Quadratic probing:* If `i` is the hash location then the following sequence of table locations is tested:

    ```
    (i+1)%N, (i+2*2)%N, (i+3*3)%N,  (i+4*4)%N, ...
    ```

    More generally, the jth "probe" of the table is $(i + c_1 j + c_2 j^2) \mod N$ where c_1 and c_2 are constants.
  - *Secondary hashing*: when a collision occurs a second hash function is applied to compute a new table location. This is repeated until an empty location is found.
- For each of these approaches, the *find* operation follows the same sequence of locations as the *insert* operation. The key value is determined to be absent from the table only when an empty location is found.
- When using open addressing to resolve collisions, the *erase* function must mark a location as "formerly occupied". If a location is instead marked empty, *find* may fail to return elements in the table. Formerly-occupied locations may (and should) be reused, but only after the *find* operation has been run to completion.
- Problems with open addressing:
  - Slows dramatically when the table is nearly full (e.g. about 80% or higher). This is particularly problematic for linear probing.
  - Fails completely when the table is full.
  - Cost of computing new hash values.

22.12 Hash Table in STL?

- The Standard Template Library standard and implementation of hash table have been slowly evolving over many years. Unfortunately, the names "hashset" and "hashmap" were spoiled by developers anticipating the STL standard, so to avoid breaking or having name clashes with code using these early implementations...
- STL's agreed-upon standard for hash tables: `unordered_set` and `unordered_map`
- Depending on your OS/compiler, you may need to add the `-std=c++11` flag to the compile line (or other configuration tweaks) to access these more recent pieces of STL. (And this will certainly continue to evolve in future years!) Also, for many types STL has a good default hash function, so you may not always need to specify both template parameters!

22.13 Our Copycat Version: A Set As a Hash Table

- The class is templated over both the key type and the hash function type.

```
template  < class KeyType, class HashFunc >
class ds_hashset {    ...    };
```

- We use separate chaining for collision resolution. Hence the main data structure inside the class is:

```
std::vector< std::list<KeyType> > m_table;
```

- We will use automatic resizing when our table is too full. Resize is expensive of course, so similar to the automatic reallocation that occurs inside the vector `push_back` function, we at least double the size of underlying structure to ensure it is rarely needed.

22.14 Our Hash Function (as a Functor or Function Object)

- Next lecture we'll talk about "function objects" or "functors".... A functor is just a class wrapper around a function, and the function is implemented as the overloaded function call operator for the class.
- Often the programmer/designer for the program using a hash function has the best understanding of the distribution of data to be stored in the hash function. Thus, they are in the best position to define a custom hash function (if needed) for the data & application.
- Here's an example of a (generically) good hash function for STL strings, wrapped up inside of a class:

```
class hash_string_obj {
public:
  unsigned int operator() (std::string const& key) const {
    // This implementation comes from
    // http://www.partow.net/programming/hashfunctions/
    unsigned int hash = 1315423911;
    for(unsigned int i = 0; i < key.length(); i++)
      hash ^= ((hash << 5) + key[i] + (hash >> 2));
    return hash;
  }
};
```

- Once our new type containing the hash function is defined, we can create instances of our hash set object containing `std::string` by specifying the type `hash_string_obj` as the second template parameter to the declaration of a `ds_hashset`. E.g.,

```
ds_hashset<std::string, hash_string_obj> my_hashset;
```

- Alternatively, we could use function pointers as a non-type template argument.
(We don't show that syntax here!).

22.15 Hash Set Iterators

- Iterators move through the hash table in the order of the storage locations rather than the ordering imposed by (say) an `operator<`. Thus, the visiting/printing order depends on the hash function and the table size.
  - Hence the increment operators must move to the next entry in the current linked list or, if the end of the current list is reached, to the first entry in the next non-empty list.
- The declaration is nested inside the `ds_hashset` declaration in order to avoid explicitly templating the iterator over the hash function type.

- The iterator must store:
  - A pointer to the hash table it is associated with. This reflects a subtle point about types: even though the `iterator` class is declared inside the `ds_hashset`, this does not mean an iterator automatically knows about any particular `ds_hashset`.
  - The index of the current list in the hash table.
  - An iterator referencing the current location in the current list.
- Because of the way the classes are nested, the `iterator` class object must declare the `ds_hashset` class as a friend, but the reverse is unnecessary.

22.16 Implementing `begin()` and `end()`

- `begin()`: Skips over empty lists to find the first key in the table. It must tie the iterator being created to the particular `ds_hashset` object it is applied to. This is done by passing the `this` pointer to the iterator constructor.
- `end()`: Also associates the iterator with the specific table, assigns an index of `-1` (indicating it is not a normal valid index), and thus does not assign the particular list iterator.
- **Exercise:** Implement the `begin()` function.

22.17 Iterator Increment, Decrement, & Comparison Operators

- The increment operators must find the next key, either in the current list, or in the next non-empty list.
- The decrement operator must check if the iterator in the list is at the beginning and if so it must proceed to find the previous non-empty list and then find the last entry in that list. This might sound expensive, but remember that the lists should be very short.
- The comparison operators must accommodate the fact that when (at least) one of the iterators is the `end`, the internal list iterator will not have a useful value.

22.18 Insert & Find

- Computes the hash function value and then the index location.
- If the key is already in the list that is at the index location, then no changes are made to the set, but an iterator is created referencing the location of the key, a pair is returned with this iterator and `false`.
- If the key is not in the list at the index location, then the key should be inserted in the list (at the front is fine), and an iterator is created referencing the location of the newly-inserted key a pair is returned with this iterator and `true`.
- **Exercise:** Implement the `insert()` function, ignoring for now the `resize` operation.
- Find is similar to insert, computing the hash function and index, followed by a `std::find` operation.

22.19 Erase

- Two versions are implemented, one based on a key value and one based on an iterator. These are based on finding the appropriate iterator location in the appropriate list, and applying the list erase function.

22.20 Resize

- Must copy the contents of the current vector into a scratch vector, resize the current vector, and then re-insert each key into the resized vector. **Exercise:** Write `resize()`

22.21 Hash Table Iterator Invalidation

- Any insert operation invalidates *all* `ds_hashset` iterators because the insert operation could cause a resize of the table. The erase function only invalidates an iterator that references the current object.

```
#ifndef ds_hashset_h_
#define ds_hashset_h_
// The set class as a hash table instead of a binary search tree.  The
// primary external difference between ds_set and ds_hashset is that
// the iterators do not step through the hashset in any meaningful
// order.  It is just the order imposed by the hash function.
#include <iostream>
#include <list>
#include <string>
#include <vector>

// The ds_hashset is templated over both the type of key and the type
// of the hash function, a function object.
template < class KeyType, class HashFunc >
class ds_hashset {
private:
  typedef typename std::list<KeyType>::iterator hash_list_itr;

public:
  // =================================================================
  // THE ITERATOR CLASS
  // Defined as a nested class and thus is not separately templated.

  class iterator {
  public:
    friend class ds_hashset;   // allows access to private variables
  private:

    // ITERATOR REPRESENTATION
    ds_hashset* m_hs;
    int m_index;               // current index in the hash table
    hash_list_itr m_list_itr;  // current iterator at the current index

  private:
    // private constructors for use by the ds_hashset only
    iterator(ds_hashset * hs) : m_hs(hs), m_index(-1) {}
    iterator(ds_hashset* hs, int index, hash_list_itr loc)
      : m_hs(hs), m_index(index), m_list_itr(loc) {}

  public:
    // Ordinary constructors & assignment operator
    iterator() : m_hs(0), m_index(-1)  {}
    iterator(iterator const& itr)
      : m_hs(itr.m_hs), m_index(itr.m_index), m_list_itr(itr.m_list_itr) {}
    iterator&  operator=(const iterator& old) {
      m_hs = old.m_hs;
      m_index = old.m_index;
      m_list_itr = old.m_list_itr;
      return *this;
    }

    // The dereference operator need only worry about the current
    // list iterator, and does not need to check the current index.
    const KeyType& operator*() const { return *m_list_itr; }

    // The comparison operators must account for the list iterators
    // being unassigned at the end.
    friend bool operator== (const iterator& lft, const iterator& rgt)
    { return lft.m_hs == rgt.m_hs && lft.m_index == rgt.m_index &&
        (lft.m_index == -1 || lft.m_list_itr == rgt.m_list_itr); }
    friend bool operator!= (const iterator& lft, const iterator& rgt)
    { return lft.m_hs != rgt.m_hs || lft.m_index != rgt.m_index ||
        (lft.m_index != -1 && lft.m_list_itr != rgt.m_list_itr); }

    // increment and decrement
    iterator& operator++() {
      this->next();
      return *this;
    }
    iterator operator++(int) {
      iterator temp(*this);
      this->next();
      return temp;
    }
    iterator & operator--() {
      this->prev();
      return *this;
    }
    iterator operator--(int) {
      iterator temp(*this);
      this->prev();
      return temp;
    }

  private:
    // Find the next entry in the table
    void next() {
      ++ m_list_itr;  // next item in the list

      // If we are at the end of this list
      if (m_list_itr == m_hs->m_table[m_index].end()) {
        // Find the next non-empty list in the table
        for (++m_index;
             m_index < int(m_hs->m_table.size()) && m_hs->m_table[m_index].empty();
             ++m_index) {}

        // If one is found, assign the m_list_itr to the start
        if (m_index != int(m_hs->m_table.size()))
          m_list_itr = m_hs->m_table[m_index].begin();

        // Otherwise, we are at the end
        else
          m_index = -1;
      }
    }

    // Find the previous entry in the table
    void prev() {
      // If we aren't at the start of the current list, just decrement
      // the list iterator
      if (m_list_itr != m_hs->m_table[m_index].begin())
        m_list_itr -- ;

      else {
        // Otherwise, back down the table until the previous
        // non-empty list in the table is found
        for (--m_index; m_index >= 0 && m_hs->m_table[m_index].empty(); --m_index) {}

        // Go to the last entry in the list.
        m_list_itr = m_hs->m_table[m_index].begin();
        hash_list_itr p = m_list_itr; ++p;
        for (; p != m_hs->m_table[m_index].end(); ++p, ++m_list_itr) {}
      }
    }
  };
  // end of ITERATOR CLASS
  // =================================================================
```

```
private:
  // =================================================================
  // HASH SET REPRESENTATION
  std::vector< std::list<KeyType> > m_table;  // actual table
  HashFunc m_hash;                            // hash function
  unsigned int m_size;                        // number of keys

public:
  // =================================================================
  // HASH SET IMPLEMENTATION

  // Constructor for the table accepts the size of the table.  Default
  // constructor for the hash function object is implicitly used.
  ds_hashset(unsigned int init_size = 10) : m_table(init_size), m_size(0) {}

  // Copy constructor just uses the member function copy constructors.
  ds_hashset(const ds_hashset<KeyType, HashFunc>& old)
    : m_table(old.m_table), m_size(old.m_size) {}

  ~ds_hashset() {}

  ds_hashset& operator=(const ds_hashset<KeyType,HashFunc>& old) {
    if (&old != this)
      *this = old;
  }

  unsigned int size() const { return m_size; }


  // Insert the key if it is not already there.
  std::pair< iterator, bool > insert(KeyType const& key) {
    const float LOAD_FRACTION_FOR_RESIZE = 1.25;

    if (m_size >= LOAD_FRACTION_FOR_RESIZE * m_table.size())
      this->resize_table(2*m_table.size()+1);

    // implemented in lecture or lab

  }

  // Find the key, using hash function, indexing and list find
  iterator find(const KeyType& key) {
    unsigned int hash_value = m_hash(key);
    unsigned int index = hash_value % m_table.size();
    hash_list_itr p = std::find(m_table[index].begin(),
                                m_table[index].end(), key);
    if (p == m_table[index].end())
      return this->end();
    else
      return iterator(this, index, p);
  }

  // Erase the key
  int erase(const KeyType& key) {
    // Find the key and use the erase iterator function.
    iterator p = find(key);
    if (p == end())
      return 0;
    else {
      erase(p);
      return 1;
    }
  }

  // Erase at the iterator
  void erase(iterator p) {
    m_table[ p.m_index ].erase(p.m_list_itr);
  }

  // Find the first entry in the table and create an associated iterator
  iterator begin() {
    // implemented in lecture or lab

  }

  // Create an end iterator.
  iterator end() {
    iterator p(this);
    p.m_index = -1;
    return p;
  }

  // A public print utility.
  void print(std::ostream & ostr) {
    for (unsigned int i=0; i<m_table.size(); ++i) {
      ostr << i << ": ";
      for (hash_list_itr p = m_table[i].begin(); p != m_table[i].end(); ++p)
        ostr << ' ' << *p;
      ostr << std::endl;
    }
  }

private:
  // resize the table with the same values but a
  void resize_table(unsigned int new_size) {
    // implemented in lecture or lab

  }
};
#endif
```

CSCI-1200 Data Structures — Spring 2017
Lecture 23 – Functors & Hash Tables, part II

Review from Lecture 22

- Hash Tables, Hash Functions, and Collision Resolution
- Performance of: Hash Tables vs. Binary Search Trees
- Collision resolution: separate chaining vs open addressing
- STL's `unordered_set` (and `unordered_map`)

Today's Lecture

- Using STL's `for_each`
- Something weird & cool in C++... Function Objects, a.k.a. *Functors*
- Continuing with Hash Tables...
  – STL's `unordered_set` (and `unordered_map`)
  – Using a hash table to implement a set/map
  – Hash functions as functors/function objects
  – Iterators, find, insert, and erase

23.1 Using STL's `for_each`

- First, here's a tiny helper function:

```
void float_print (float f) {
  std::cout << f << std::endl;
}
```

- Let's make an STL vector of floats:

```
std::vector<float> my_data;
my_data.push_back(3.14);
my_data.push_back(1.41);
my_data.push_back(6.02);
my_data.push_back(2.71);
```

- Now we can write a loop to print out all the data in our vector:

```
std::vector<float>::iterator itr;
for (itr = my_data.begin(); itr != my_data.end(); itr++) {
  float_print(*itr);
}
```

- Alternatively we can use it with STL's `for_each` function to visit and print each element:

```
std::for_each(my_data.begin(), my_data.end(), float_print);
```

  Wow! That's a lot less to type. Can I stop using regular `for` and `while` loops altogether?

- We can actually also do the same thing without creating & explicitly naming the `float_print` function. We create an *anonymous function* using *lambda*:

```
std::for_each(my_data.begin(), my_data.end(), [](float f){ std::cout << f << std::end; });
```

  Lambda is new to the C++ language (part of C++11). But lambda is a core piece of many classic, older programming languages including Lisp and Scheme. Python lambdas and Perl anonymous subroutines are similar. (In fact lambda dates back to the 1930's, before the first computers were built!) You'll learn more about lambda more in later courses like CSCI 4430 Programming Languages!

23.2 Function Objects, a.k.a. *Functors*

- In addition to the basic mathematical operators `+ - * / < >` , another operator we can overload for our C++ classes is the *function call operator*.

  Why do we want to do this? This allows instances or objects of our class, to be used like functions. It's weird but powerful.

- Here's the basic syntax. Any specific number of arguments can be used.

```
class my_class_name {
public:
  // ... normal class stuff ...
  my_return_type operator() ( /* my list of args */ );
};
```

23.3 Why are Functors Useful?

- One example is the default 3rd argument for `std::sort`. We know that by default STL's sort routines will use the less than comparison function for the type stored inside the container. How exactly do they do that?
- First let's define another tiny helper function:

```
bool float_less(float x, float y) {
  return x < y;
}
```

- Remember how we can sort the `my_data` vector defined above using our own homemade comparison function for sorting:

```
std::sort(my_data.begin(),my_data.end(),float_less);
```

  If we don't specify a 3rd argument:

```
std::sort(my_data.begin(),my_data.end());
```

  This is what STL does by default:

```
std::sort(my_data.begin(),my_data.end(),std::less<float>());
```

- What is `std::less`? It's a templated class. Above we have called the default constructor to make an instance of that class. Then, that instance/object can be used like it's a function. Weird!
- How does it do that? `std::less` is a teeny tiny class that just contains the overloaded function call operator.

```
template <class T>
class less {
public:
  bool operator() (const T& x, const T& y) const { return x < y; }
};
```

  You can use this instance/object/functor as a function that expects exactly two arguments of type `T` (in this example `float`) that returns a bool. That's exactly what we need for `std::sort`! This ultimately does the same thing as our tiny helper homemade compare function!

23.4 Another more Complicated Functor Example

- Constructors of function objects can be used to specify *internal data* for the functor that can then be used during computation of the function call operator! For example:

```
class between_values {
private:
  float low, high;
public:
  between_values(float l, float h) : low(l), high(h) {}
  bool operator() (float val) { return low <= val && val <= high; }
};
```

- The range between `low` & `high` is specified when a functor/an instance of this class is created. We might have multiple different instances of the `between_values` functor, each with their own range. Later, when the functor is used, the query value will be passed in as an argument. The function call operator accepts that single argument `val` and compares against the internal data `low` & `high`.
- This can be used in combination with STL's `find_if` construct. For example:

```
between_values two_and_four(2,4);

if (std::find_if(my_data.begin(), my_data.end(), two_and_four) != my_data.end()) {
  std::cout << "Found a value greater than 2 & less than 4!" << std::endl;
}
```

- Alternatively, we could create the functor without giving it a variable name. And in the use below we also capture the return value to print out the first item in the vector inside this range. Note that it does not print all values in the range.

```
std::vector<float>::iterator itr;
itr = std::find_if(my_data.begin(), my_data.end(), between_values(2,4));
if (itr != my_data.end()) {
  std::cout << "my_data contains " << *itr
            << ", a value greater than 2 & less than 4!" << std::endl;
}
```

23.5 Using STL's Associative Hash Table (Map)

- Using the default `std::string` hash function.
  - With no specified initial table size.
    `std::unordered_map<std::string,Foo> m;`
  - Optionally specifying initial (minimum) table size.
    `std::unordered_map<std::string,Foo> m(1000);`
- Using a home-made `std::string` hash function. Note: We are required to specify the initial table size.
  - Manually specifying the hash function type.
    `std::unordered_map<std::string,Foo,std::function<unsigned int(std::string)> > m(1000, MyHashFunction);`
  - Using the `decltype` specifier to get the "declared type of an entity".
    `std::unordered_map<std::string,Foo,decltype(&MyHashFunction)> m(1000, MyHashFunction);`
- Using a a home-made `std::string` hash functor or function object.
  - With no specified initial table size.
    `std::unordered_map<std::string,Foo,MyHashFunctor> m;`
  - Optionally specifying initial (minimum) table size.
    `std::unordered_map<std::string,Foo,MyHashFunctor> m(1000);`
- Note: In the above examples we're creating a association between two types (STL strings and custom `Foo` object). If you'd like to just create a set (no associated 2nd type), simply switch from `unordered_map` to `unordered_set` and remove the `Foo` from the template type in the examples above.

CSCI-1200 Data Structures — Spring 2017
Lecture 24 – Priority Queues

Review from Lectures 22 & 23

- Hash Tables, Hash Functions, and Collision Resolution
- Performance of: Hash Tables vs. Binary Search Trees
- Collision resolution: separate chaining vs open addressing
- STL's `unordered_set` (and `unordered_map`)
- Using a hash table to implement a set/map
  - Hash functions as functors/function objects
  - Iterators, find, insert, and erase
- Using STL's `for_each`
- Something weird & cool in C++... Function Objects, a.k.a. *Functors*

Today's Lecture

- STL Queue and STL Stack
- Definition of a Binary Heap
- What's a Priority Queue?
- A Priority Queue as a Heap
- A Heap as a Vector
- Building a Heap
- Heap Sort
- If time allows... Merging heaps are the motivation for *leftist heaps*

24.1 Additional STL Container Classes: Stacks and Queues

- We've studied STL vectors, lists, maps, and sets. These data structures provide a wide range of flexibility in terms of operations. One way to obtain computational efficiency is to consider a simplified set of operations or functionality.
- For example, with a hash table we give up the notion of a sorted table and gain in find, insert, & erase efficiency.
- 2 additional examples are:
  - **Stacks** allow access, insertion and deletion from only one end called the *top*
    - There is no access to values in the middle of a stack.
    - Stacks may be implemented efficiently in terms of vectors and lists, although vectors are preferable.
    - All stack operations are $O(1)$
  - **Queues** allow insertion at one end, called the *back* and removal from the other end, called the *front*
    - There is no access to values in the middle of a queue.
    - Queues may be implemented efficiently in terms of a list. Using vectors for queues is also possible, but requires more work to get right.
    - All queue operations are $O(1)$

24.2 Suggested Exercises: Tree Traversal using a Stack and Queue

Given a pointer to the root node in a binary tree:

- Use an STL `stack` to print the elements with a pre-order traversal ordering. *This is straightforward.*
- Use an STL `stack` to print the elements with an in-order traversal ordering. *This is more complicated.*
- Use an STL `queue` to print the elements with a breadth-first traversal ordering.

24.3 What's a Priority Queue?

- Priority queues are used in prioritizing operations. Examples include a personal "to do" list, what order to do homework assignments, jobs on a shop floor, packet routing in a network, scheduling in an operating system, or events in a simulation.
- Among the data structures we have studied, their interface is most similar to a queue, including the idea of a `front` or `top` and a `tail` or a `back`.
- Each item is stored in a priority queue using an associated "priority" and therefore, the `top` item is the one with the lowest value of the priority score. The `tail` or `back` is never accessed through the public interface to a priority queue.
- The main operations are `insert` or `push`, and `pop` (or `delete_min`).

24.4 Some Data Structure Options for Implementing a Priority Queue

- Vector or list, either sorted or unsorted
  - At least one of the operations, `push` or `pop`, will cost linear time, at least if we think of the container as a linear structure.
- Binary search trees
  - If we use the priority as a `key`, then we can use a combination of finding the minimum key and erase to implement `pop`. An ordinary binary-search-tree insert may be used to implement `push`.
  - This costs logarithmic time in the average case (and in the worst case as well if balancing is used).
- The latter is the better solution, but we would like to improve upon it — for example, it might be more natural if the minimum priority value were stored at the root.
  - We will achieve this with binary *heap*, giving up the complete ordering imposed in the binary *search tree*.

24.5 Definition: Binary Heaps

- A binary heap is a complete binary tree such that at each internal node, p, the value stored is less than the value stored at either of p's children.
  - A complete binary tree is one that is completely filled, except perhaps at the lowest level, and at the lowest level all leaf nodes are as far to the left as possible.
- Binary heaps will be drawn as binary trees, but implemented **using vectors**!
- Alternatively, the heap could be organized such that the value stored at each internal node is greater than the values at its children.

24.6 Exercise: Drawing Binary Heaps

Draw two different binary heaps with these values: 52 13 48 7 32 40 18 25 4

Draw several other trees with these values that *not* binary heaps.

24.7 Implementing Pop (a.k.a. Delete Min)

- The value at the top (root) of the tree is replaced by the value stored in the last leaf node.
  This has echoes of the erase function in binary search trees.
- The last leaf node is removed.
  QUESTION: But how do we find the last leaf? Ignore this for now...
- The value now at the root likely breaks the heap property. We use the `percolate_down` function to restore the heap property. This function is written here in terms of tree nodes with child pointers (and the priority stored as a `value`), but later it will be written in terms of vector subscripts.

```
percolate_down(TreeNode<T> * p) {
  while (p->left) {
    TreeNode<T>* child;
    //  Choose the child to compare against
    if (p->right && p->right->value < p->left->value)
      child = p->right;
    else
      child = p->left;
    if (child->value < p->value) {
      swap(child, p); // value and other non-pointer member vars
      p = child;
    }
    else
      break;
  }
}
```

24.8 Implementing Push (a.k.a. Insert)

- To add a value to the heap, a new last leaf node in the tree is created to store that value.
- Then the `percolate_up` function is run. It assumes each node has a pointer to its parent.

```
percolate_up(TreeNode<T> * p) {
  while (p->parent)
    if (p->value < p->parent->value) {
      swap(p, parent);  // value and other non-pointer member vars
      p = p->parent;
    }
    else
      break;
}
```

24.9 Push (Insert) and Pop (Delete-Min) Usage Exercise

Suppose the following operations are applied to an initially empty binary heap of integers. Show the resulting heap after each `delete_min` operation. (Remember, the tree must be **complete**!)

```
push 5, push 3, push 8, push 10, push 1, push 6,
pop,
push 14, push 2, push 4, push 7,
pop,
pop,
pop
```

24.10 Heap Operations Analysis

- Both `percolate_down` and `percolate_up` are $O(\log n)$ in the worst-case. Why?
- But, `percolate_up` (and as a result `push`) is $O(1)$ in the average case. Why?

24.11 Implementing a Heap with a Vector (instead of Nodes & Pointers)

- In the vector implementation, the tree is never explicitly constructed. Instead the heap is stored as a vector, and the child and parent "pointers" can be implicitly calculated.
- To do this, number the nodes in the tree starting with 0 first by level (top to bottom) and then scanning across each row (left to right). These are the vector indices. Place the values in a vector in this order.
- As a result, for each subscript, i,
  - The parent, if it exists, is at location $\lfloor (i-1)/2 \rfloor$.
  - The left child, if it exists, is at location $2i+1$.
  - The right child, if it exists, is at location $2i+2$.
- For a binary heap containing n values, the last leaf is at location $n-1$ in the vector and the last internal (non-leaf) node is at location $\lfloor (n-1)/2 \rfloor$.
- The standard library (STL) `priority_queue` is implemented as a binary heap.

24.12 Heap as a Vector Exercises

- Draw a binary heap with values: 52 13 48 7 32 40 18 25 4, first as a tree of nodes & pointers, then in vector representation.
- Starting with an initially empty heap, show the vector contents for the binary heap after each `delete_min` operation.

```
push 8, push 12, push 7, push 5, push 17, push 1,
pop,
push 6, push 22, push 14, push 9,
pop,
pop,
```

24.13 Building A Heap

- In order to build a heap from a vector of values, for each index from $\lfloor (n-1)/2 \rfloor$ down to 0, run `percolate_down`. Show that this fully organizes the data as a heap and requires at most $O(n)$ operations.
- If instead, we ran `percolate_up` from each index starting at index 0 through index n-1, we would get properly organized heap data, but incur a $O(n \log n)$ cost. Why?

24.14 Heap Sort

- Heap Sort is a simple algorithm to sort a vector of values: Build a heap and then run n consecutive `pop` operations, storing each "popped" value in a new vector.
- It is straightforward to show that this requires $O(n \log n)$ time.
- **Exercise:** Implement an *in-place* heap sort. An in-place algorithm uses only the memory holding the input data – a separate large temporary vector is not needed.

24.15 Summary Notes about Vector-Based Priority Queues

- Priority queues are conceptually similar to queues, but the order in which values / entries are removed ("popped") depends on a priority.
- Heaps, which are conceptually a binary tree but are implemented in a vector, are the data structure of choice for a priority queue.
- In some applications, the priority of an entry may change while the entry is in the priority queue. This requires that there be "hooks" (usually in the form of indices) into the internal structure of the priority queue. This is an implementation detail we have not discussed.

CSCI-1200 Data Structures — Spring 2017
Lecture 25 — C++ Inheritance and Polymorphism

Review from Lecture 24

- STL Queues and STL Stacks
- Definition of a Binary Heap
- What's a Priority Queue?
- A Priority Queue as a Heap
- A Heap as a Vector
- Building a Heap
- Heap Sort

Today's Class

- Inheritance is a relationship among classes. Examples: bank accounts, polygons, stack & list
- Basic mechanisms of inheritance
- Types of inheritance
- Is-A, Has-A, As-A relationships among classes.
- Polymorphism

25.1 Motivating Example: Bank Accounts

- Consider different types of bank accounts:
  - Savings accounts
  - Checking accounts
  - Time withdrawal accounts (like savings accounts, except that only the interest can be withdrawn)
- If you were designing C++ classes to represent each of these, what member functions might be repeated among the different classes? What member functions would be unique to a given class?
- To avoid repeating common member functions and member variables, we will create a **class hierarchy**, where the common members are placed in a **base class** and specialized members are placed in **derived classes**.

25.2 Accounts Hierarchy

- `Account` is the *base class* of the hierarchy.
- `SavingsAccount` is a *derived* class from `Account`. `SavingsAccount` has inherited member variables & functions and ordinarily-defined member variables & functions.
- The member variable `balance` in base class `Account` is `protected`, which means:
  - `balance` is NOT publicly accessible outside the class, but it is accessible in the derived classes.
  - if `balance` was declared as `private`, then `SavingsAccount` member functions could not access it.
- When using objects of type `SavingsAccount`, the inherited and derived members are treated exactly the same and are not distinguishable.
- `CheckingAccount` is also a derived class from base class `Account`.
- `TimeAccount` is derived from `SavingsAccount`. `SavingsAccount` is its base class and `Account` is its indirect base class.

25.3 Exercise: Draw the Accounts Class Hierarchy

```
#include <iostream>
// Note we've inlined all the functions (even though some are > 1 line of code)

class Account {
public:
  Account(double bal = 0.0) : balance(bal) {}
  void deposit(double amt) { balance += amt; }
  double get_balance() const { return balance; }
protected:
  double balance;   // account balance
};

class SavingsAccount : public Account {
public:
  SavingsAccount(double bal = 0.0, double pct = 5.0)
    : Account(bal), rate(pct/100.0) {}
  double compound() { // computes and deposits interest
    double interest = balance * rate;
    balance += interest;
    return interest;
  }
  double withdraw(double amt) { // if overdraft ==> return 0, else return amount
    if (amt > balance) {
      return 0.0;
    } else {
      balance -= amt;
      return amt;
    }
  }
protected:
  double rate;  // periodic interest rate
};

class CheckingAccount :  public Account {
public:
  CheckingAccount(double bal = 0.0, double lim = 500.0, double chg = 0.5)
     : Account(bal), limit(lim), charge(chg) {}
  double cash_check(double amt) {
    assert (amt > 0);
    if (balance < limit && (amt + charge <= balance)) {
      balance -= amt + charge;
      return amt + charge;
    } else if (balance >= limit && amt <= balance) {
      balance -= amt;
      return amt;
    } else {
      return 0.0;
    }
  }
protected:
  double limit;    // lower limit for free checking
  double charge;   // per check charge
};

class TimeAccount : public SavingsAccount {
public:
  TimeAccount(double bal = 0.0, double pct = 5.0)
    : SavingsAccount(bal, pct), funds_avail(0.0) {}
  // redefines 2 member functions from SavingsAccount
  double compound() {
    double interest = SavingsAccount::compound();
    funds_avail += interest;
    return interest;
  }
```

```
  double withdraw(double amt) {
    if (amt <= funds_avail) {
      funds_avail -= amt;
      balance -= amt;
      return amt;
    } else {
      return 0.0;
    }
  }
  double get_avail() const { return funds_avail; };
protected:
  double funds_avail;  // amount available for withdrawal
};
```

25.4 Constructors and Destructors

- Constructors of a derived class *call the base class constructor* immediately, before doing ANYTHING else. The only thing you can control is which constructor is called and what the arguments will be. Thus when a `TimeAccount` is created 3 constructors are called: the `Account` constructor, then the `SavingsAccount` constructor, and then finally the `TimeAccount` constructor.
- The reverse is true for destructors: derived class constructors do their jobs first and then base class destructors are called at the, automatically. *Note: destructors for classes which have derived classes must be marked virtual for this chain of calls to happen.*

25.5 Overriding Member Functions in Derived Classes

- A derived class can redefine member functions in the base class. The function prototype must be identical, not even the use of `const` can be different (otherwise both functions will be accessible).
- For example, see `TimeAccount::compound` and `TimeAccount::withdraw`.
- Once a function is redefined it is not possible to call the base class function, unless it is explicitly called as in `SavingsAccount::compound`.

25.6 Public, Private and Protected Inheritance

- Notice the line `class Savings_Account : public Account {`

  This specifies that the member functions and variables from `Account` do not change their *public*, *protected* or *private* status in `SavingsAccount`. This is called *public* inheritance.
- *protected* and *private* inheritance are other options:
  - With protected inheritance, public members becomes protected and other members are unchanged
  - With private inheritance, all members become private.

25.7 Stack Inheriting from List

- For another example of inheritance, let's re-implement the `stack` class as a derived class of `std::list`:

```
template <class T>
class stack : private std::list<T> {
public:
  stack() {}
  stack(stack<T> const& other) : std::list<T>(other) {}
  virtual ~stack() {}
  void push(T const& value) { this->push_back(value); }
  void pop() { this->pop_back(); }
  T const& top() const { return this->back(); }
  int size() { return std::list<T>::size(); }
  bool empty() { return std::list<T>::empty(); }
};
```

- Private inheritance hides the `std::list<T>` member functions from the outside world. However, these member functions are still available to the member functions of the `stack<T>` class.
- Note: no member variables are defined — the only member variables needed are in the list class.

- When the stack member function uses the same name as the base class (list) member function, the name of the base class followed by :: must be provided to indicate that the base class member function is to be used.
- The copy constructor just uses the copy constructor of the base class, without any special designation because the stack object is a list object as well.

25.8 Is-A, Has-A, As-A Relationships Among Classes

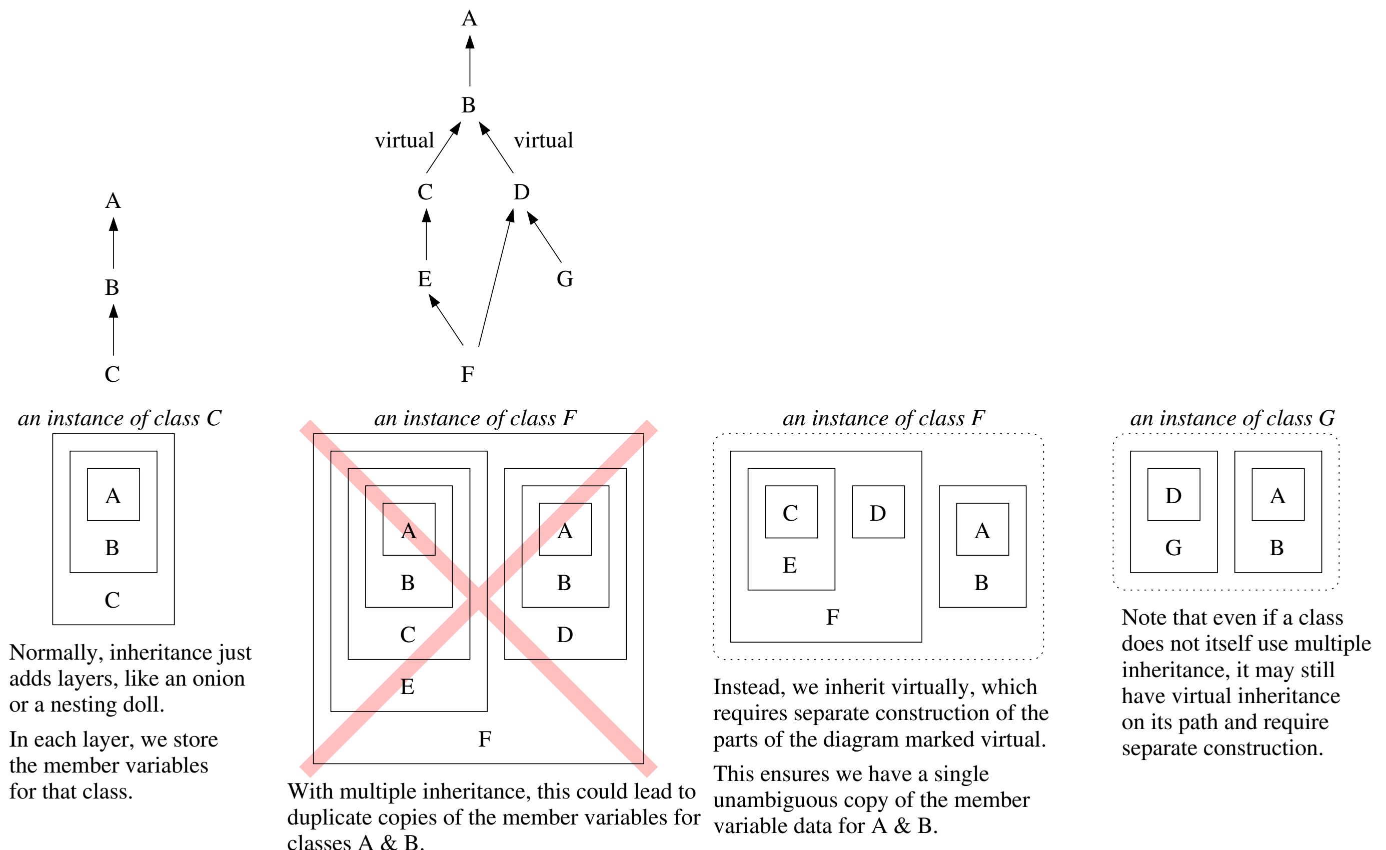
- When trying to determine the relationship between (hypothetical) classes C1 and C2, try to think of a logical relationship between them that can be written:
  - C1 is a C2,
  - C1 has a C2, or
  - C1 is implemented as a C2
- If writing "C1 is-a C2" is best, for example: "a savings account is an account", then C1 should be a derived class (a subclass) of C2.
- If writing "C1 has-a C2" is best, for example: "a cylinder has a circle as its base", then class C1 should have a member variable of type C2.
- In the case of "C1 is implemented as-a C2", for example: "the stack is implemented as a list", then C1 should be derived from C2, but with private inheritance. This is by far the least common case!

25.9 Exercise: 2D Geometric Primitives

Create a class hierarchy of geometric objects, such as: triangle, isosceles triangle, right triangle, quadrilateral, square, rhombus, kite, trapezoid, circle, ellipse, etc. How should this hierarchy be arranged? What member variables and member functions should be in each class?

25.10 Note: Multiple Inheritance

- When sketching a class hierarchy for geometric objects, your may have wanted to specify relationships that were more complex... in particular some objects may wish to inherit from *more than one base class.*
- This is called *multiple inheritance* and can make many implementation details significantly more hairy. Different programming languages offer different variations of multiple inheritance.



25.11 Introduction to Polymorphism

- Let's consider a small class hierarchy version of polygonal objects:

```
class Polygon {
public:
  Polygon() {}
  virtual ~Polygon() {}
  int NumVerts() { return verts.size(); }
  virtual double Area() = 0;
  virtual bool IsSquare() { return false; }
protected:
  vector<Point> verts;
};

class Triangle : public Polygon {
public:
  Triangle(Point pts[3]) {
    for (int i = 0; i < 3; i++) verts.push_back(pts[i]); }
  double Area();
};

class Quadrilateral : public Polygon {
public:
  Quadrilateral(Point pts[4]) {
    for (int i = 0; i < 4; i++) verts.push_back(pts[i]); }
  double Area();
  double LongerDiagonal();
  bool IsSquare() { return (SidesEqual() && AnglesEqual()); }
private:
  bool SidesEqual();
  bool AnglesEqual();
};
```

- Functions that are common, at least have a common interface, are in `Polygon`.
- Some of these functions are marked `virtual`, which means that when they are redefined by a derived class, this new definition will be used, even for pointers to base class objects.
- Some of these virtual functions, those whose declarations are followed by `= 0` are *pure virtual*, which means they must be redefined in a derived class.
  - Any class that has pure virtual functions is called "abstract".
  - Objects of abstract types may not be created — only pointers to these objects may be created.
- Functions that are specific to a particular object type are declared in the derived class prototype.

25.12 A Polymorphic List of Polygon Objects

- Now instead of two separate lists of polygon objects, we can create one "polymorphic" list:

```
std::list<Polygon*> polygons;
```

- Objects are constructed using `new` and inserted into the list:

```
Polygon *p_ptr = new Triangle( .... );
polygons.push_back(p_ptr);
p_ptr = new Quadrilateral( ... );
polygons.push_back(p_ptr);
Triangle *t_ptr = new Triangle( .... );
polygons.push_back(t_ptr);
```

Note: We've used the same pointer variable (`p_ptr`) to point to objects of two different types.

25.13 Accessing Objects Through a Polymorphic List of Pointers

- Let's sum the areas of all the polygons:

  ```
  double area = 0;
  for (std::list<Polygon*>::iterator i = polygons.begin(); i!=polygons.end(); ++i)
     area += (*i)->Area();
  ```

  Which `Area` function is called? If `*i` points to a `Triangle` object then the function defined in the `Triangle` class would be called. If `*i` points to a `Quadrilateral` object then `Quadrilateral::Area` will be called.

- Here's code to count the number of squares in the list:

  ```
  int count = 0;
  for (std::list<Polygon*>::iterator i = polygons.begin(); i!=polygons.end(); ++i)
     count += (*i)->IsSquare();
  ```

  If `Polygon::IsSquare` had not been declared `virtual` then the function defined in `Polygon` would always be called! In general, given a pointer to type `T` we start at `T` and look "up" the hierarchy for the closest function definition (this can be done at compile time). If that function has been declared `virtual`, we will start this search instead at the actual type of the object (this requires additional work at runtime) in case it has been redefined in a derived class of type `T`.

- To use a function in `Quadrilateral` that is not declared in `Polygon`, you must "cast" the pointer. The pointer `*q` will be NULL if `*i` is not a `Quadrilateral` object.

  ```
  for (std::list<Polygon*>::iterator i = polygons.begin(); i!=polygons.end(); ++i) {
     Quadrilateral *q = dynamic_cast<Quadrilateral*> (*i);
     if (q) std::cout << "diagonal: " <<  q->LongerDiagonal() << std::endl;
  }
  ```

25.14 Exercise

What is the output of the following program?

```
class Base {
 public:
  Base() {}
  virtual void A() { std::cout << "Base A  "; }
  void B() { std::cout << "Base B  "; }
};

class One : public Base {
public:
  One() {}
  void A() { std::cout << "One A  "; }
  void B() { std::cout << "One B  "; }
};
class Two : public Base {
public:
  Two() {}
  void A() { std::cout << "Two A  "; }
  void B() { std::cout << "Two B  "; }
};

int main() {
  Base* a[3];
  a[0] = new Base;
  a[1] = new One;
  a[2] = new Two;
  for (unsigned int i=0; i<3; ++i) {
    a[i]->A();
    a[i]->B();
  }
  std::cout << std::endl;
  return 0;
}
```

CSCI-1200 Data Structures — Spring 2017
Lecture 26 — C++ Exceptions

Review from Lecture 25

- Inheritance is a relationship among classes. Examples: bank accounts, polygons, stack & list
- Basic mechanisms of inheritance
- Types of inheritance
- Is-A, Has-A, As-A relationships among classes.
- Polymorphism

Today's Class

- Error handling strategies
- Basic exception mechanisms: `try`/`throw`/`catch`
- Functions & exceptions, constructors & exceptions
- STL `exceptions`
- RAII "Resource Acquisition is Initialization"
- Structured Exception Handling in the Windows Operating System
- Google's C++ Style Guide on Exceptions
- Some examples from today's lecture are drawn from:
  http://www.cplusplus.com/doc/tutorial/exceptions/
  http://www.parashift.com/c++-faq-lite/exceptions.html

26.1 Error Handling Strategy A: Optimism (a.k.a. Naivety or Denial)

- **Assume there are no errors.** Command line arguments will always be proper, any specified files will always be available for read/write, the data in the files will be formatted correctly, numerical calculations will not attempt to divide by zero, etc.

```
double answer = numer / denom;
```

- For small programs, for short term use, by a single programmer, where the input is well known and controlled, this may not be a disaster (and is often fastest to develop and thus a good choice).
- But for large programs, this code will be challenging to maintain. It can be difficult to pinpoint the source of an error. The symptom of a problem (if noticed at all) may be many steps removed from the source. The software system maintainer must be familiar with the assumptions of the code (which is difficult if there is a ton of code, the code was written some time ago, by someone else, or is not sufficiently commented... or all of the above!).

26.2 Error Handling Strategy B: Plan for the Worst Case (a.k.a. Paranoia)

- Anticipate every mistake or source of error (or as many as you can think of). **Write lots of `if` statements everywhere there may be a problem**. Write code for what to do instead, print out error messages, and/or exit when nothing seems reasonable.

```
double answer;
// for some application specific epsilon (often not easy to specify)
double epsilon = 0.00001;
if (fabs(denom) < epsilon) {
  std::cerr << "detected a divide by zero error" << std::endl;
  // what to do now?  (often there is no "right" thing to do)
  answer = 0;
} else {
  answer = numer / denom;
}
```

- Error checking & error handling generally requires a lot of programmer time to write all of this error code.
- The code gets bulkier and harder to understand/maintain.
- If a nested function call might have a problem, and the error needs to be propagated back up to a function much earlier on the call stack, all the functions in between must also test for the error condition and pass the error along. (This is messy to code and all that error checking has performance implications).
- Creating a comprehensive test suite (yes, error checking/handling code must be tested too!) that exercises all the error cases is extremely time consuming, and some error situations are very difficult to produce.

26.3 Error Handling Strategy C: If/When It Happens We'll Fix It (a.k.a. Procrastination)

- Again, anticipate everything that might go wrong and just **call `assert` in lots of places**. This can be somewhat less work than the previous option (we don't need to decide what to do if the error happens, the program just exits immediately).

```
double epsilon = 0.00001;
assert (fabs(denom) > epsilon);
answer = numer / denom;
```

- This can be a great tool during the software development process. Write code to test all (or most) of the assumptions in each function/code unit. Quickly get a prototype system up and running that works for the general, most common, non-error cases first.
- If/when an unexpected input or condition occurs, then additional code can be written to more appropriately handle special cases and errors.
- However, the use of assertions is generally frowned upon in real-world production code (users don't like to receive seemingly arbitrary & total system failures, especially when they paid for the software!).
- Once you have completed testing & debugging, and are fairly confident that the likely error cases are appropriately handled, then the `gcc` compile flag `-DNDEBUG` flag can be used to remove all remaining `assert` statements before compiling the code (conveniently removing any performance overhead for assert checking).

26.4 Error Handling Strategy D: The Elegant Industrial-Strength Solution

- **Use exceptions.** Somewhat similar to Strategy B, but in practice, code written using exceptions results in more efficient code (and less overall code!) and that code is less prone to programming mistakes.

```
double epsilon = 0.00001;
try {
  if (fabs(denom) < epsilon) {
    throw std::string("divide by zero");
  }
  double answer = numer / denom;
  /* do lots of other interesting work here with the answer! */
}
catch (std::string &error) {
  std::cerr << "detected a " << error << " error" << std::endl;
  /* what to do in the event of an error */
}
```

26.5 Basic Exception Mechanisms: Throw

- When you detect an error, `throw` an exception. Some examples:

```
throw 20;
throw std::string("hello");
throw Foo(2,5);
```

- You can throw a value of any type (e.g., `int`, `std::string`, an instance of a custom class, etc.)
- When the throw statement is triggered, the rest of that block of code is abandoned.

26.6 Basic Exception Mechanisms: Try/Catch

- If you suspect that a fragment of code you are about to execute may throw an exception and you want to prevent the program from crashing, you should wrap that fragment within a try/catch block:

```
try {
  /* the code that might throw */
}
catch (int x) {
  /* what to do if the throw happened
     (may use the variable x)
  */
}
/* the rest of the program */
```

- The logic of the try block may throw more than one type of exception.
- A catch statement specifies what type of exception it catches (e.g., `int`, `std::string`, etc.)
- You may use multiple catch blocks to catch different types of exceptions from the same try block.
- You may use `catch (...) { /* code */ }` to catch *all* types of exceptions. (But you don't get to use the value that was thrown!)
- If an exception is thrown, the program searches for the closest *enclosing* try/catch block with the appropriate type. That try/catch may be several functions away on the call stack (it might be all the way back in the main function!).
- If no appropriate catch statement is found, the program exits, e.g.:

```
terminate called after throwing an instance of 'bool'
Abort trap
```

26.7 Basic Exception Mechanisms: Functions

- If a function you are writing might throw an exception, you can specify the type of exception(s) in the prototype.

```
int my_func(int a, int b) throw(double,bool) {
  if (a > b)
    throw 20.3;
  else
    throw false;
}

int main() {
  try {
    my_func(1,2);
  }
  catch (double x) {
    std::cout << " caught a double " << x << std::endl;
  }
  catch (...) {
    std::cout << " caught some other type " << std::endl;
  }
}
```

- If you use the throw syntax in the prototype, and the function throws an exception of a type that you have not listed, the program will terminate immediately (it can't be caught by any enclosing try statements).
- If you don't use the throw syntax in the prototype, the function may throw exceptions of any type, and they may be caught by an appropriate try/catch block.

26.8 Comparing Method B (explicit if tests) to Method D (exceptions)

- Here's code using exceptions to sort a collection of lines by slope:

```
class Point {
public:
  Point(double x_, double y_) : x(x_),y(y_) {}
  double x,y;
};

class Line {
public:
  Line(const Point &a_, const Point &b_) : a(a_),b(b_) {}
  Point a,b;
};

double compute_slope(const Point &a, const Point &b) throws(int) {
  double rise = b.y - a.y;
  double run = b.x - a.x;
  double epsilon = 0.00001;
  if (fabs(run) < epsilon) throw -1;
  return rise / run;
}

double slope(const Line &ln) {
  return compute_slope(ln.a,ln.b);
}

bool steeper_slope(const Line &m, const Line &n) {
  double slope_m = slope(m);
  double slope_n = slope(n);
  return slope_m > slope_n;
}

void organize(std::vector<Line> &lines) {
  std::sort(lines.begin(),lines.end(), steeper_slope);
}

int main () {
  std::vector<Line> lines;
  /* omitting code to initialize some data */
  try {
    organize(lines);
    /* omitting code to print out the results */
  } catch (int) {
    std::cout << "error: infinite slope" << std::endl;
  }
}
```

- Specifically note the behavior if one of the lines has infinite slope (a vertical line).
- Note also how the exception propagates out through several nested function calls.
- **Exercise:** Rewrite this code to have the same behavior but *without exceptions*. Try to preserve the overall structure of the code as much as possible. (Hmm... it's messy!)

26.9 STL `exception` Class

- STL provides a base class `std::exception` in the `<exception>` header file. You can derive your own exception type from the exception class, and overwrite the `what()` member function

```
class myexception: public std::exception {
  virtual const char* what() const throw() {
    return "My exception happened";
  }
};

int main () {
  myexception myex;
  try {
    throw myex;
  }
  catch (std::exception& e) {
    std::cout << e.what() << std::endl;
  }
  return 0;
}
```

- The STL library throws several different types of exceptions (all derived from the STL `exception` class):

`bad_alloc`	thrown by new on allocation failure
`bad_cast`	thrown by `dynamic_cast` (when casting to a reference variable rather than a pointer)
`bad_exception`	thrown when an exception type doesn't match any catch
`bad_typeid`	thrown by typeid
`ios_base::failure`	thrown by functions in the iostream library

26.10 Exceptions & Constructors

- The only way for a constructor to fail is to throw an exception.
- A common reason that a constructor must fail is due to a failure to allocate memory. If the system cannot allocate sufficient memory resources for the object, the `bad_alloc` exception is thrown.

```
  try {
    int* myarray= new int[1000];
  }
  catch (std::exception& e) {
    std::cout << "Standard exception: " << e.what() << std::endl;
  }
```

- It can also be useful to have the constructor for a custom class throw a descriptive exception if the arguments are invalid in some way.

26.11 Resource Acquisition Is Initialization (RAII)

- Because exceptions might happen at any time, and thus cause the program to abandon a partially executed function or block of code, it may not be appropriate to rely on a `delete` call that happens later on in a block of code.
- RAII describes a programming strategy to ensure proper deallocation of memory despite the occurrence of exceptions. The goal is to ensure that resources are released before exceptions are allowed to propagate.
- Variables allocated on the stack (not dynamically-allocated using `new`) are guaranteed to be properly destructed when the variable goes out of scope (e.g., when an exception is thrown and we abandon a partially executed block of code or function).
- Special care must be taken for dynamically-allocated variables (and other resources like open files, mutexes, etc.) to ensure that the code is *exception safe*.

26.12 Structured Exception Handing (SEH) in the Windows Operating System

- The Windows Operating System has special language support, called Structured Exception Handing (SEH), to handle hardware exceptions. Some examples of hardware exceptions include divide by zero and segmentation faults (there are others!).
- In Unix/Linux/Mac OSX these hardware exceptions are instead dealt with using signal handlers. Unfortunately, writing error handling code using signal handlers incurs a larger performance hit (due to `setjmp`) and the design of the error handling code is less elegant than the usual C++ exception system because signal handlers are global entities.

26.13 Google's C++ Style Guide on Exceptions

https://google.github.io/styleguide/cppguide.html#Exceptions

Pros:

- Exceptions allow higher levels of an application to decide how to handle "can't happen" failures in deeply nested functions, without the obscuring and error-prone bookkeeping of error codes.
- Exceptions are used by most other modern languages. Using them in C++ would make it more consistent with Python, Java, and the C++ that others are familiar with.
- Some third-party C++ libraries use exceptions, and turning them off internally makes it harder to integrate with those libraries.
- Exceptions are the only way for a constructor to fail. We can simulate this with a factory function or an Init() method, but these require heap allocation or a new "invalid" state, respectively.
- Exceptions are really handy in testing frameworks.

Cons:

- When you add a throw statement to an existing function, you must examine all of its transitive callers. Either they must make at least the basic exception safety guarantee, or they must never catch the exception and be happy with the program terminating as a result. For instance, if f() calls g() calls h(), and h throws an exception that f catches, g has to be careful or it may not clean up properly.
- More generally, exceptions make the control flow of programs difficult to evaluate by looking at code: functions may return in places you don't expect. This causes maintainability and debugging difficulties. You can minimize this cost via some rules on how and where exceptions can be used, but at the cost of more that a developer needs to know and understand.
- Exception safety requires both RAII and different coding practices. Lots of supporting machinery is needed to make writing correct exception-safe code easy. Further, to avoid requiring readers to understand the entire call graph, exception-safe code must isolate logic that writes to persistent state into a "commit" phase. This will have both benefits and costs (perhaps where you're forced to obfuscate code to isolate the commit). Allowing exceptions would force us to always pay those costs even when they're not worth it.
- Turning on exceptions adds data to each binary produced, increasing compile time (probably slightly) and possibly increasing address space pressure.
- The availability of exceptions may encourage developers to throw them when they are not appropriate or recover from them when it's not safe to do so. For example, invalid user input should not cause exceptions to be thrown. We would need to make the style guide even longer to document these restrictions!

Decision:

On their face, the benefits of using exceptions outweigh the costs, especially in new projects. However, for existing code, the introduction of exceptions has implications on all dependent code. If exceptions can be propagated beyond a new project, it also becomes problematic to integrate the new project into existing exception-free code. Because most existing C++ code at Google is not prepared to deal with exceptions, it is comparatively difficult to adopt new code that generates exceptions.

Given that Google's existing code is not exception-tolerant, the costs of using exceptions are somewhat greater than the costs in a new project. The conversion process would be slow and error-prone. We don't believe that the available alternatives to exceptions, such as error codes and assertions, introduce a significant burden.

Our advice against using exceptions is not predicated on philosophical or moral grounds, but practical ones. Because we'd like to use our open-source projects at Google and it's difficult to do so if those projects use exceptions, we need to advise against exceptions in Google open-source projects as well. Things would probably be different if we had to do it all over again from scratch.

There is an exception to this rule (no pun intended) for Windows code.

CSCI-1200 Data Structures — Spring 2017
Lecture 27 — Garbage Collection & Smart Pointers

Announcements

- Please fill out your course evaluations!
- Those of you interested in becoming an undergraduate mentor for Data Structures, or another CSCI course:
  - Speak to your graduate lab TA and ask him/her to recommend you for the position.
  - A week or two before the start of the Spring term, David Goldschmidt will post the online application for mentors for CS1, DS, and other CSCI courses. He'll send it to the CSCI undergraduate mailing list, but it will (probably) also be posted on Facebook & Reddit.
- The final exam pratice problems will be posted on the calendar this afternoon.
  - If we get at least 85% response to the course evaluations, we will post the solutions early.

Review from Lecture 26

- Error handling strategies
- Basic exception mechanisms: `try/throw/catch`
- Functions & exceptions, constructors & exceptions

Today's Lecture

- What is Garbage?
- 3 Garbage Collection Techniques
- Smart Pointers

27.1 What is Garbage?

- Not everything sitting in memory is useful. Garbage is anything that cannot have any influence on the future computation.
- With C++, the programmer is expected to perform *explicit memory management*. You must use `delete` when you are done with dynamically allocated memory (which was created with `new`).
- In Java, and other languages with "garbage collection", you are not required to explicitly de-allocate the memory. The system automatically determines what is garbage and returns it to the available pool of memory.

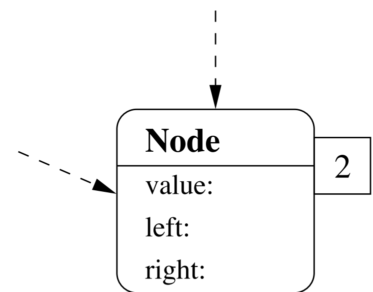
  Certainly this makes it easier to learn to program in these languages, but *automatic memory management* does have performance and memory usage disadvantages.
- Today we'll overview 3 basic techniques for automatic memory management.

27.2 The Node class

- For our discussion today, we'll assume that all program data is stored in dynamically-allocated instances of the following simple class. This class can be used to build linked lists, trees, and graphs with cycles:

```
class Node {
public:
  Node(char v, Node* l, Node* r) :
    value(v), left(l), right(r) {}
  char value;
  Node* left;
  Node* right;
};
```

27.3 Garbage Collection Technique #1: Reference Counting



1. Attach a *counter* to each `Node` in memory.
2. When a new pointer is connected to that `Node`, increment the counter.
3. When a pointer is removed, decrement the counter.
4. Any `Node` with `counter == 0` is garbage and is available for reuse.

27.4 Reference Counting Exercise

- Draw a "box and pointer" diagram for the following example, keeping a "reference counter" with each `Node`.

```
Node *a = new Node('a', NULL, NULL);
Node *b = new Node('b', NULL, NULL);
Node *c = new Node('c', a, b);
a = NULL;
b = NULL;
c->left = c;
c = NULL;
```

- Is there any garbage?

27.5 Memory Model Exercise

- In memory, we pack the `Node` instances into a big array. In the toy example below, we have only enough room in memory to store 8 `Node`s, which are addressed 100 $\rightarrow$ 107. 0 is a NULL address.
- For simplicity, we'll assume that the program uses only one variable, `root`, through which it accesses all of the data. Draw the box-and-pointer diagram for the data accessible from `root` = 105.

address	100	101	102	103	104	105	106	107
value	a	b	c	d	e	f	g	h
left	0	0	100	100	0	102	105	104
right	0	100	103	0	105	106	0	0

```
root:  105
```

- What memory is garbage?

27.6 Garbage Collection Technique #2: Stop and Copy

1. Split memory in half (*working memory* and *copy memory*).
2. When out of working memory, stop computation and begin garbage collection.
   (a) Place `scan` and `free` pointers at the start of the copy memory.
   (b) Copy the `root` to copy memory, incrementing `free`. Whenever a node is copied from working memory, leave a *forwarding address* to its new location in copy memory in the left address slot of its old location.
   (c) Starting at the `scan` pointer, process the left and right pointers of each node. Look for their locations in working memory. If the node has already been copied (i.e., it has a forwarding address), update the reference. Otherwise, copy the location (as before) and update the reference.
   (d) Repeat until scan == free.
   (e) Swap the roles of the working and copy memory.

27.7 Stop and Copy Exercise

Perform stop-and-copy on the following with `root` = 105:

WORKING MEMORY

address	100	101	102	103	104	105	106	107
value	a	b	c	d	e	f	g	h
left	0	0	100	100	0	102	105	104
right	0	100	103	0	105	106	0	0

COPY MEMORY

address	108	109	110	111	112	113	114	115
value								
left								
right								

```
root:  105
scan:
free:
```

27.8 Garbage Collection Technique #3: Mark-Sweep

1. Add a mark bit to each location in memory.
2. Keep a free pointer to the head of the free list.
3. When memory runs out, stop computation, clear the mark bits and begin garbage collection.
4. Mark
   (a) Start at the `root` and follow the accessible structure (keeping a *stack* of where you still need to go).
   (b) Mark every node you visit.
   (c) Stop when you see a marked node, so you don't go into a cycle.
5. Sweep
   (a) Start at the end of memory, and build a new free list.
   (b) If a node is unmarked, then it's garbage, so hook it into the free list by chaining the left pointers.

27.9 Mark-Sweep Exercise

Let's perform Mark-Sweep on the following with `root` = 105:

```
address   100  101  102  103  104  105  106  107
value       a    b    c    d    e    f    g    h
left        0    0  100  100    0  102  105  104
right       0  100  103    0  105  106    0    0
marks


root:  105
free:
stack:
```

27.10 Garbage Collection Comparison

- **Reference Counting:**
  - \+ fast and incremental
  - – can't handle cyclical data structures!
  - ? requires ∼33% extra memory (1 integer per node)
- **Stop & Copy:**
  - – requires a long pause in program execution
  - \+ can handle cyclical data structures!
  - – requires 100% extra memory (you can only use half the memory)
  - \+ runs fast if most of the memory is garbage (it only touches the nodes reachable from the root)
  - \+ data is clustered together and memory is "de-fragmented"
- **Mark-Sweep:**
  - – requires a long pause in program execution
  - \+ can handle cyclical data structures!
  - \+ requires ∼1% extra memory (just one bit per node)
  - – runs the same speed regardless of how much of memory is garbage.
    It must touch all nodes in the mark phase, and must link together all garbage nodes into a free list.

27.11 Practical Garbage Collection Methodology in C++: Smart Pointers

- Garbage collection looks like an attractive option both when we are quickly drafting a prototype system and also when we are developing big complex programs that process and rearrange lots of data.
- Unfortunately, general-purpose, invisible garbage collection isn't something we can just tack onto C++, an enormous beast of a programming language (but that doesn't stop people from trying!). So is there anything we can do? Yes, we can use *Smart Pointers* to gain some of the features of garbage collection.
- Some examples below are modified from these nice online references:
  http://ootips.org/yonat/4dev/smart-pointers.html
  http://www.codeproject.com/KB/stl/boostsmartptr.aspx
  http://en.wikipedia.org/wiki/Smart_pointer
  http://www.boost.org/doc/libs/1_48_0/libs/smart_ptr/smart_ptr.htm

27.12 What's a Smart Pointer?

- The goal is to create a widget that works just like a regular pointer most of the time, except at the beginning and end of its lifetime. The syntax of how we construct smart pointers is a bit different and we don't need to obsess about how & when it will get deleted (it happens automatically).
- Here's one flavor of a smart pointer (much simplified from STL):

```
template <class T>
class auto_ptr {
public:
  explicit auto_ptr(T* p = NULL) : ptr(p) {}  /* prevents cast/conversion */
  ~auto_ptr()      { delete ptr; }
  T& operator*()   { return *ptr; }
  T* operator->()  { return ptr; }            /* fakes being a pointer */
private:
  T* ptr;
};
```

- And let's start with some example code without smart pointers:

```
void foo() {
  Polygon* p(new Polygon(/* stuff */));
  p->DoSomething();
  delete p;
}
```

- Here's how we can re-write the same example with our `auto_ptr`:

```
void foo() {
  auto_ptr<Polygon> p(new Polygon(/* stuff */);
  p->DoSomething();
}
```

- We don't have to call delete! There's no memory leak or memory error in this code. Awesome!

27.13 So, What are the Advantages of Smart Pointers?

- Smart pointers are magical. They allow us to be lazy! All the time we spent learning about dynamically allocated memory, copy constructors, destructors, memory leaks, and segmentation faults this semester was unnecessary. *Whoa... that's overstating things more than slightly!!*
- With practice, smart pointers can result in code that is more concise and elegant with fewer errors. *Why? ...*
- With thoughtful use, smart pointers make it easier to follow the principles of RAII and make code *exception safe*. In the `auto_ptr` example above, if `DoSomething` throws an exception, the memory for object `p` will be properly deallocated when we leave the scope of the `foo` function! This is *not* the case with the original version.
- The STL `shared_ptr` flavor implements reference counting garbage collection. Awesome^2!
- They play nice with STL containers. Say you make an `std::vector` (or `std::list`, or `std::map`, etc.) of regular pointers to Polygon objects, `Polygon*` (especially handy if this is a polymorphic collection of objects!). You allocate them all with `new`, and when you are all finished you must remember to *explicitly* deallocate each of the objects.

```
class Polygon { /*...*/ };
class Triangle : public Polygon { /*...*/ };
class Quad : public Polygon { /*...*/ };

std::vector<Polygon*> polys;
polys.push_back(new Triangle(/*...*/));
polys.push_back(new Quad(/*...*/));

for (unsigned int i = 0; i < polys.size(); i++) {
  delete polys[i];
}
polys.clear();
```

In contrast with smart pointers they will be deallocated automagically!

```
std::vector<shared_ptr<Polygon> > polys;

polys.push_back(shared_ptr<Polygon>(new Triangle(/*...*/)));
polys.push_back(shared_ptr<Polygon>(new Quad(/*...*/)));

polys.clear();  // cleanup is automatic!
```

27.14 Why are Smart Pointers Tricky?

- Smart pointers **do not alleviate the need to master pointers, basic memory allocation & deallocation, copy constructors, destructors, assignment operators, and reference variables**.
- You can still make mistakes in your smart pointer code that yield the same types of memory corruption, segmentation faults, and memory leaks as regular pointers.
- There are several different flavors of smart pointers to choose from (developed for different uses, for common *design patterns*). You need to understand your application *and* the different pitfalls when you select the appropriate implementation.

27.15 What are the Different Types of Smart Pointers?

Like other parts of the C++ standard, these tools are still evolving. The different choices reflect different *ownership semantics* and different *design patterns*. There are some smart pointers in STL, and also some in Boost (a C++ library that further extends the current STL). A quick overview:

- `auto_ptr`
  When "copied" (copy constructor), the new object takes ownership and the old object is now empty. *Deprecated in new C++ standard.*
- `unique_ptr`
  Cannot be copied (copy constructor not public). Can only be "moved" to transfer ownership. Explicit ownership transfer. *Intended to replace* `auto_ptr`. std::unique_ptr has memory overhead only if you provide it with some non-trivial deleter. It has time overhead only during constructor (if it has to copy the provided deleter) and during destructor (to destroy the owned object).
- `scoped_ptr` (Boost)
  "Remembers" to delete things when they go out of scope. Alternate to `auto_ptr`. Cannot be copied.
- `shared_ptr`
  Reference counted ownership of pointer. Unfortunately, circular references are still a problem. Different sub-flavors based on where the counter is stored in memory relative to the object, e.g., `intrusive_ptr`, which is more memory efficient. std::unique_ptr has memory overhead only if you provide it with some non-trivial deleter. It has time overhead in constructor (to create the reference counter), in destructor (to decrement the reference counter and possibly destroy the object) and in assignment operator (to increment the reference counter).
- `weak_ptr`
  Use with `shared_ptr`. Memory is destroyed when no more `shared_ptr`s are pointing to object. So each time a `weak_ptr` is used you should first "lock" the data by creating a `shared_ptr`.
- `scoped_array` and `shared_array` (Boost)

CSCI-1200 Data Structures — Spring 2017
Lecture 28 — Concurrency & Asynchronous Computing

Final Exam General Information

- The final exam will be held: **Wednesday May 10th from 3-6pm. Your room and zone assignment will be posted on the homework server next week.**

  A makeup exam will only be offered if required by the RPI rules regarding final exam conflicts *-OR-* if a written excuse from the Dean of Students office is provided. Contact the *ds_instructors* list by email immediately if you have a conflict.

- Coverage: Lectures 1-28, Labs 1-14, and HW 1-10.
- Closed-book and closed-notes *except for 2 sheets of 8.5x11 inch paper (front & back) that may be handwritten or printed*. Computers, cell-phones, music players, and other electronic equipment are not permitted and must be turned off.
- **All students must bring their Rensselaer photo ID card.**
- The best thing you can do to prepare for the final is practice. Try the review problems (posted on the course website) with pencil & paper first. Then practice programming (with a computer) the exercises and other exercises from lecture, lab, homework and the textbook. Solutions to the review problems will be posted several days before the final exam.
- Please check the homework submission server data entry for your grades early next week. Email your lab TA if there is any error before the final exam.

Review from Lecture 27 & Lab

- What is garbage? Memory which cannot (or should not) be accessed by the program and is available for reuse.
- Explicit memory management (C++) vs. automatic garbage collection.
- Reference Counting, Stop & Copy, Mark-Sweep.
- Cyclical data structures, memory overhead, incremental vs. pause in execution, ratio of good to garbage, defragmentation.
- Smart Pointers

28.1 Today's Class

- Computing with multiple threads/processes and one or more processors
- Shared resources & mutexes/locks
- Deadlock: the Dining Philosopher's Problem

28.2 The Role of Time in Evaluation

- Sometimes the order of evaluation does matter, and sometimes it doesn't.
  - The behavior of objects with *state* depends on sequence of events that have occurred.
  - *Referential transparency*: when equivalent expressions can be substituted for one another without changing the value of the expression. For example, a complex expression can be replaced with its result *if* repeated evaluations always yield the same result, independent of context.
- What happens when objects don't change one at a time but rather act concurrently?
  - We may be able to take advantage of this by letting threads/processes run at the same time (a.k.a., in parallel).
  - However, we will need to think carefully about the interactions and shared resources.

28.3 Concurrency Example: Joint Bank Account

- Consider the following bank account implementation:

```
class Account {
public:
  Account(int amount) : balance(amount) {}
  void deposit(int amount) {
    int tmp = balance;                              // A
    tmp += amount;                                  // B
    balance = tmp;                                  // C
  }
  void withdraw(int amount) {
    int tmp = balance;                              // D
    if (amount > tmp)
      cout << "Error: Insufficient Funds!" << endl;   // E1
    else {
      tmp -= amount;                                // E2
    }
    balance = tmp;                                  // F
  }
private:
  int balance;
};
```

- We create a joint account that will be used by two people (threads/processes):

```
Account account(100);
```

- Now, enumerate all of the possible interleavings of the sub-expressions (`A-F`) if the following two function calls were to happen concurrently. What are the different outcomes?

```
account.deposit(50);
account.withdraw(125);
```

- What if instead the actions were:

```
account.deposit(50);
account.withdraw(75);
```

28.4 Correct/Acceptable Behavior of Concurrent Programs

- No two operations that change any shared state variables may occur at the same time.
  - Certain low-level operations are guaranteed to execute *atomic*-ly (from start to finish without interruption), but this varies based on the hardware and operating system. We need to know which operations are *atomic* on our hardware.
  - In the bank account example we *cannot* assume that the `deposit` and `withdraw` functions are atomic.
- The concurrent system should produce the same result as if the threads/processes had run sequentially *in some order*.
  - We do not require that the threads/processes run sequentially, only that they produce results as if they had run sequentially.
  - *Note:* There may be more than one correct result!
- **Exercise:** What are the acceptable outcomes for the bank account example?

28.5 Serialization via a Mutex

- We can *serialize* the important interactions using a primitive, atomic synchronization method called a *mutex*.
- Once one thread has acquired the mutex (locking the resource), no other thread can acquire the mutex until it has been released.
- In the example below we use the STL `mutex` object (`#include <mutex>`). If the mutex is unavailable, the call to the `mutex` member function `lock()` *blocks* (the thread pauses at that line of code until the mutex is available).

```
class Chalkboard {
public:
  Chalkboard() { }
  void write(Drawing d) {
    board.lock();
    drawing = d;
    board.unlock();
  }
  Drawing read() {
    board.lock();
    Drawing answer = drawing;
    board.unlock();
    return answer;
  }
private:
  Drawing drawing;
  std::mutex board;
};
```

- What does the mutex do in this code?

28.6 The Professor & Student Classes

- Here are two simple classes that can communicate through a shared `Chalkboard` object:

```
class Professor {
public:
  Professor(Chalkboard *c) { chalkboard = c; }
  virtual void Lecture(const std::string &notes) {
    chalkboard->write(notes);
  }
protected:
  Chalkboard* chalkboard;
};


class Student {
public:
  Student(Chalkboard *c) { chalkboard = c; }
  void TakeNotes() {
    Drawing d = chalkboard->read();
    notebook.push_back(d);
  }
private:
  Chalkboard* chalkboard;
  std::vector<Drawing> notebook;
};
```

28.7 Launching Concurrent Threads

- So how exactly do we get multiple streams of computation happening simultaneously? There are many choices (may depend on your programming language, operating system, compiler, etc.).
- We'll use the STL `thread` library (`#include <thread>`). The new thread begins execution in the provided function (`student_thread`, in this example). We pass the necessary shared data from the main thread to the secondary thread to facilitate communication.

```
#define num_notes 10

void student_thread(Chalkboard *chalkboard) {
  Student student(chalkboard);
  for (int i = 0; i < num_notes; i++) {
    student.TakeNotes();
  }
}

int main() {
  Chalkboard chalkboard;
  Professor prof(&chalkboard);
  std::thread student(student_thread, &chalkboard);
  for (int i = 0; i < num_notes; i++) {
    prof.Lecture("blah blah");
  }
  student.join();
}
```

- The `join` command pauses to wait for the secondary thread to finish computation before continuing with the program (or exiting in this example).
- What can still go wrong? How can we fix it?

28.8 Condition Variables

- Here we've added a *condition variable*, `student_done`:

```
class Chalkboard {
public:
  Chalkboard() { student_done = true; }
  void write(Drawing d) {
    while (1) {
      board.lock();
      if (student_done) {
        drawing = d;
        student_done = false;
        board.unlock();
        return;
      }
      board.unlock();
    }
  }
  Drawing read() {
    while (1) {
      board.lock();
      if (!student_done) {
        Drawing answer = drawing;
        student_done = true;
        board.unlock();
        return answer;
      }
      board.unlock();
    }
  }
```

```
private:
  Drawing drawing;
  std::mutex board;
  bool student_done;
};
```

- *Note:* This implementation is actually quite inefficient due to "busy waiting". A better solution is to use a operating system-supported *condition variable* that yields to other threads if the lock is not available and is signaled when the lock becomes available again. STL has a `condition_variable` type which allows you to `wait` for or `notify` other threads that it may be time to resume computation.

28.9 Exercise: Multiple Students and/or Multiple Professors

- Now consider that we have multiple students and/or multiple professors. How can you ensure that each student is able to copy a complete set of notes?

28.10 Multiple Locks & Deadlock

- For this last example, we add two public member variables of type `std::mutex` to the `Chalkboard` class, named `chalk` and `textbook`.
- And we derive two different types of lecturer from the base class `Professor`. The professors can lecture concurrently, but they must share the chalk and the book.

```
class CautiousLecturer : public Professor {
public:
  CautiousLecturer(Chalkboard *c) : Professor(c) {}
  void Lecture() {
    chalkboard->textbook.lock();
    Drawing d = FromBookDrawing();
    chalkboard->chalk.lock();
    Professor::Lecture(d);
    chalkboard->chalk.unlock();
    chalkboard->textbook.unlock();
  }
};

void checkDrawing(const Drawing &d) {}

class BrashLecturer : public Professor {
public:
  BrashLecturer(Chalkboard *c) : Professor(c) {}
  void Lecture() {
    chalkboard->chalk.lock();
    Drawing d = FromMemoryDrawing();
    Professor::Lecture(d);
    chalkboard->textbook.lock();
    checkDrawing(d);
    chalkboard->textbook.unlock();
    chalkboard->chalk.unlock();
  }
};
```

- What can go wrong? How can we fix it?
  Why might philosophers discuss this problem over dinner?

28.11 Topics Covered

- Algorithm analysis: big O notation; best case, average case, or worst case; algorithm running time or additional memory usage
- STL classes: `string`, `vector`, `list`, `map`, & `set`, (we talked about but did not practice using STL `stack`, `queue`, `unordered_set`, `unordered_map`, & `priority_queue`)
- C++ Classes: constructors (default, copy, & custom argument), assignment operator, & destructor, classes with dynamically-allocated memory, operator overloading, inheritance, polymorphism
- Subscripting (random-access, pointer arithmetic) vs. iteration
- Recursion & problem solving techniques
- Memory: pointers & arrays, heap vs. stack, dynamic allocation & deallocation of memory, garbage collection, smart pointers
- Implementing data structures: resizable arrays (vectors), linked lists (singly-linked, doubly-linked, circularly-linked, dummy head/tail nodes), trees (for sets & maps), hash sets
- Binary Search Trees, tree traversal (in-order, pre-order, post-order, depth-first, & breadth-first)
- Hash tables (hash functions, collision resolution), priority queues, heap as a vector
- Exceptions, concurrency & asynchronous computing

28.12 Course Summary

- Approach any problem by studying the requirements carefully, playing with hand-generated examples to understand them, and then looking for analogous problems that you already know how to solve.
- STL offers container classes and algorithms that simplify the programming process and raise your conceptual level of thinking in designing solutions to programming problems. Just think how much harder some of the homework problems would have been without generic container classes!
- When choosing between algorithms and between container classes (data structures) you should consider:
  - efficiency,
  - naturalness of use, and
  - ease of programming.
- Use classes with well-designed public and private member functions to encapsulate sections of code.
- Writing your own container classes and data structures usually requires building linked structures and managing memory through the big three:
  - copy constructor,
  - assignment operator, and
  - destructor.
- When testing and debugging:
  - Test one function and one class at a time,
  - Figure out what your program actually does, not what you wanted it to do,
  - Use small examples and boundary conditions when testing, and
  - Find and fix the first mistake in the flow of your program before considering other apparent mistakes.
- Above all, remember the excitement and satisfaction when your hard work and focused debugging is rewarded with a program that demonstrates your technical mastery and realizes your creative problem solving skills!