CS32: Data Structures and Algorithms

Professor Smallberg

Thilan Tran

Winter 2018

Contents

CS32: Data Structures and Algorithms 4

Linking C++ Files 4
- Linking and Multiple Compilation . . . 4
  - Header Files . . . 7
  - Include Guards . . . 8
  - Circular Dependency . . . 9
- Constructing with Member Initialization Lists . . . 11

Pointers & Dynamic Arrays 13
- Writing a String type . . . 13
- Default Parameter Values . . . 16
- Copy Constructors & Assignment Operator Overloading . . . 17
  - Copy Construction . . . 19
  - Assignment Operator . . . 19
- Inline Methods . . . 20

Data Structures and Algorithms 20
- Linked Lists . . . 20
  - Possible Approaches . . . 21
  - General Tips . . . 22
  - Implementation . . . 22
- Stacks and Queues . . . 23
  - Stacks . . . 23
  - Queues . . . 24
  - Priority Queues . . . 24
  - Syntax . . . 25

Stack Implementations . . . 26
Queue Implementations . . . 26
Evaluating Expressions with Algorithms . . . 27
Evaluating Postfix Expression . . . 27
Translating Infix to Postfix . . . 28
Trees . . . 28
Example . . . 29
Binary Trees . . . 30
Binary Search Tree . . . 31
Hash Table . . . 32
Dynamic / Resizable Hash Table . . . 33
Hash Functions . . . 34
Heap . . . 35
Tables . . . 37
Graphs . . . 37

Standard Template Library (STL) 37
Vectors . . . 38
Implementation . . . 39
List . . . 39
Set . . . 40
Map . . . 41
Auto Keyword . . . 42
Iterators . . . 43
Iterators with Templates . . . 45
STL Algorithms . . . 46

Recursion 48
Recursion Examples . . . 49
Sorting Algorithm . . . 49
Finding a Target . . . 49
Solving a Maze . . . 50

Big-O 51
Operations with Arrays . . . 51
More Examples . . . 52
Process for Evaulating Big-O . . . 52
Multiple Variables . . . 53
STL Cheat Sheet . . . 53

Sorting Algorithms 54
Selection Sort . . . 54
Bubble Sort . . . 55

Shell Sort . . . 56
Insertion Sort . . . 56
Merge Sort . . . 57
Quick Sort . . . 58
Heap Sort . . . 60
Stability of Sorts . . . 61

Inheritance **62**
Drawing with Shapes . . . 62
Inheritance . . . 65
Inheritance in Memory . . . 66
Functions within Base and Derived Classes . . . 66
Virtual Functions . . . 66
Overriding Functions . . . 67
Virtual Functions Demystified . . . 68
Pure Virtual Functions . . . 69
Construction . . . 69
Destruction . . . 70

Function and Class Templates **71**
Function Templates . . . 71
Useful Construction Syntax . . . 73
Template Errors . . . 73
Class Templates . . . 74
Template Specialization . . . 76

Appendix **76**
File Input / Output . . . 76
Output . . . 76
Input . . . 76
Stream Parameters . . . 78
Object Oriented Design Steps . . . 79

Discussion **80**
Administrative . . . 80
1.18.19 . . . 80
2.1.19 . . . 81

CS32: Data Structures and Algorithms

Linking C++ Files

Linking and Multiple Compilation

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

// Circle.h "Header File"

class Circle
{
  public:
    Circle(double x, double y, double r);
    bool scale(double factor);
    void draw() const;
    double radius() const; // member function promises not to modify the object
                           // always consider if functions should be declared const

    // now let's try and make sure data can never be in a bad state
    // always document restrictions / invariants

    // Class invariant:
    //   m_r > 0 (check boundary cases)

  private:
    double m_x;
    double m_y;
    double m_r;
};

// double area(Circle x);      // non-member function, passing by value, x is a copy
// double area(Circle& x);     // passing by reference,
                               // x is another name for an existing Circle, cheaper
double area(const Circle& x); // passing by constant reference, x is another name,
```

```
                                // promises not to modify x, cheaper
    // still won't compile, area() calls radius() which is not a const function!

// now let's implement Circle class and area()
// interface (what) vs. implementation (how)

// Circle.cpp "Implementation File"

#include "Circle.h"
#include <iostream>
#include <cstdlib>
#include <string>
#include <math>
using namespace std;

Circle::Circle(double x, double y, double r)
  : m_x(x), m_y(y), m_r(r) // member initialization list, executes before function body
{
  if (r <= 0) // how do we deal with this? constructors don't have a return value
              // can't use default value, constructor should not return normally
              // if constructor doesn't create a meaningful object,
              // could use exceptions in this case
  {
    cerr << "Cannot create a circle with radius " << r << endl;
    exit(1);
  }
  // m_x = x;
  // m_y = y;
  // m_r = r;
}

bool Circle::scale(double factor)
{
  if (factor <= 0) // how do we deal with issue? can return a bool
                   // check boundary! we decided circle of radius 0 is invalid
    return false;
  m_r *= factor;
  return true;
}

void Circle::draw() const
{
```

```
  ...
}

double Circle::radius() const
{
  return m_r;
}

double area(Circle x)
{
  const double PI = 4 * atan(1.0);
  return PI * x.radius() * x.radius();
}

// let's say above code and main routine are on separate files
// how to get it to compile?

// myapp.cpp

#include "Circle.h" // double quotes references an actual file somewhere
#include <iostream> // angle brackets references some predefined directories
                    // ie. compiler specifications
using namespace std;

int main()
{
  Circle c(8, -3, 3.7);
  Circle d(-2, 5, 10);
  d.scale(2);      // some people argue to always test for valid return values
  // d.m_r = -10; // how do we prevent this? public / private distinction
                  // now won't compile, main() isn't a member function
                  // but how can area() access the m_r member now?
                  // use a public accessor function
  // language alternative: private variables were accessible but not modifiable?
  // then it's very difficult to change implementations! principle of abstraction
  d.draw();
  cout << area(d);
  double e;
  cin >> e;
  if(!d.scale(e)) // now we can check if scale() was valid
    ...
}
```

- normally, compiler takes c++ source code (.cpp), translates to an object file (.obj)
- object file is still not executable, example code only has code for the main routine
  - object file defines the main routine
  - object file needs the implementations for
    - Circle::Circle()
    - Circle::scale()
    - Circle::draw()
    - area()
    - operator `<<`
  - some of these implementations are found in the library code: defines `<<` , other standard library functions
- object file must be linked with the library code to generate an executable (linker)
- if Circle's functions still aren't implemented, building will generate a linker error, **not** a compilation error
  - implementing functions more than once is another linker error
- preferably, split large programs across multiple files
  - compiling and linking many object files is better than compiling one large file (*separate compilation*)
  - will reuse different up-to-date object files instead of recompiling and then relink (*relinking faster than recompilation*)

Header Files

- in example code, compiling the two files creates two object files that satisfy each other
  - but we would have had to declare the entire Circle class **twice**
  - use *header files*!
    - for every class, convention is to have a header file and implementation file
    - #include "Header.h" acts as if entire header file code has been copied here, shorthand
    - don't have to explicitly compile header files!
      - eg. command line compilation: g32 -o myapp myapp.cpp Circle.cpp
      - or g32 -o myapp *.cpp
      - using * as a wildcard
- **never** #include .cpp files!
  - can lead to multiple defined functions

Include Guards

```
// Point.h
// =======

#ifndef POINT_INCLUDED // "include guard"
#define POINT_INCLUDED
class Point
{
  ...
};
#endif  // POINT_INCLUDED

// Circle.h
// ========

#ifndef CIRCLE_INCLUDED
#define CIRCLE_INCLUDED

#include "Point.h"

class Circle
{
  ...
  Point m_center;
  double m_r;
}
#endif  // CIRCLE_INCLUDED

// myapp.cpp
// =========

// #include "Point.h" // bad rule, necessitates including both header files to
                      // use Circle class, and Point.h before Circle.h
#include "Circle.h"
// #include "Point.h" // this would lead to a multiple defined linking error!
                      // declares Point class twice
#include "Point.h"    // this is fine after adding include guards

int main()
{
  Circle c(...);
  Point p(...);
```

```
  ...
}
```

- As a general rule, header files should **always** include other header files they need to compile
- but could lead to errors when including both Circle.h and Point.h
  - can use *include guards* with #ifndef, #endif
  - include guard is not necessary when multiple .cpp files include the same file
  - every header's include guard name should have different names
  - convention of #define HEADER_INCLUDED

Circular Dependency

```
// Student.h
// =========

#ifndef STUDENT_INCLUDED
#define STUDENT_INCLUDED

// #include "Course.h" // needs to know what a course is, circular dependency

class Course;

class Student
{
  ...
  void enroll(Cource* cp);
  ...
  Course* m_studyList[10];
  ...
};
#endif  // STUDENT_INCLUDED

// Course.h
// ========

#ifndef COURSE_INCLUDED
#define COURSE_INCLUDED

// #include "Student.h" // needs to know what a student is, circular dependency

class Student;
```

```
class Course
{
  ...
  Student* m_roster[1000];
  ...
};
#endif  // COURSE_INCLUDED

// blah.cpp
// ========

#include "Student.h"
#include "Course.h"

void f(Student* sp, Course* cp)
{
  sp->enroll(cp);
}

// foo.cpp  fine to have multiple includes and incomplete type declarations
//=========

#include <iostream>
#include <iostream>
#include <iostream>
class A;
class A;
class A // but only one actual declaration of class A
{
  ...
};
```

- *circular dependency*! will not resolve by including headers
- the compiler just needs to know how big the class declarations are
  - but all pointers to class types are the **same** size
  - thus, don't need to know all the details of a class to use a pointer to that class
    - or to use a class as a *parameter or a return type*
  - so convention is to use incomplete type declarations whenever possible (**cheaper**)
    - avoid unnecessary #include's
  - as soon as details of a class are necessary, the header file **must** be in-

cluded
  * eg. using member functions, constructors, etc.

Constructing with Member Initialization Lists

```
class Circle
{
  public:
    Circle(double x, double y, double r);
    ...
  private:
    double m_x;
    double m_y;
    double m_r;
};

Circle::Circle(double x, double y, double r);
  : m_x(x), m_y(y)
{
  m_r = r;
}

class StickFigure
{
  public:
    StickFigure(double bl, double headDiam, std::string nm, double hx, double hy);
  private:
    std::string m_name;
    Circle m_head;
    double m_bodyLength;
};

StickFigure:: StickFigure(double bl, double headDiam, std::string nm, double hx, double hy)
  : m_name(nm), m_head(hx, hy, headDiam/2), m_bodyLength(bl)
  // must initialize m_head here, m_name is optional to initialize here
  // good practice to initialize everything here
{
  // m_name = nm; // first empty string then assignment, slightly more expensive
  // m_x = hx;    // private members of Circle! can't use simple assignment
  // m_y = hy;
  // m_r = headDiam / 2;
  // m_head = ??? // already previously constructed with default constructor
```

```
                 // (won't compile, didn't exist)
  // m_bodyLength = bl;
  ...
}

struct Employee
{
  string name;
  double salary;
  int age;
};

Employee e;

// ====== compiler-written constructor:
// Employee::Employee()
// {}

Circle c(-2, 5, 10);
// first m_x and m_y are initialized in member initialization list,
// then m_r is updated in constructor body

StickFigure sf(5, 4, "Fred", -2, 7);
// m_name first empty string, then "Fred"
// but if assigned using member initalizer list,
// doesn't call default constructor, cheaper
```

- construction/assignment nuances
  - for built-in types, **simple**: just change the contents
    * doesn't matter if value is assigned later in body or initialized
  - for class types, **different**:
    * whenever object of a class type is declared/created, a constructor is **always** called
    * when assigning to a class type, it's an *already* constructed object, assignment != construction
    * eg. string starts out with empty string
    * for member objects in a class, may **have** to use member initialization list to construct them with parameters
  - recall: if you declare no constructors at all, the compiler writes a zero-argument constructor (default constructor)
- steps of construction:
  1. construct the base object

2. construct each data member, consulting the *member initialization list*
   - if not listed:
     * built-in type: left uninitialized
     * class type: default-constructed (having no default constructor is an **error**)
3. execute *body of constructor*

Pointers & Dynamic Arrays

Writing a String type

```
class String
{
  public:
    // String(); // should empty string have a nullptr or a pointer to a zero byte?
                 // nullptr is cheaper/faster (more users benefit),
                 // pointer to zero byte acts like any other String
                 // (programmer benefits)

    String(const char* text = "");
    // can use a default parameter instead of a default constructor
    // can be called without arguments, default constructor!
    // default constructor means a constructor that
    // CAN be called with zero arguments

    // String(String other); // WRONG syntax for copy constructor, won't compile!
                             // cyclical, needs to call itself as a copy constructor
    String(const String& other); // correct syntax using const reference

    ~String();

    // void assign(const String& rhs);       // right hand side
    // void operator=(const String& rhs);    // we want to chain operator
    // String operator=(const String& rhs); // creates an extra copy, wasteful
    String& operator=(const String& rhs);

    void swap(String& other);      // for copy and swap idiom

  private:
      // Class invariant: // always make sure to document all cases
```

```
      //  m_text points to a dynamically allocated array of m_len+1 chars
      //  m_len ≥ 0
      //  m_len == strlen(m_text)
    // char m_text[???];   // should there be a limit on the number of characters?
    char* m_text;// now has a dynamic size
    int m_len;   // how to find end of the String? store size? append zero byte?
                 // why not both?
};

String::String(const char* text) // text is a pointer to constant chars!
{
  if (text == nullptr)// let's produce empty string if nullptr was passed
    text = "";        // text can point to different places,
                      // but can't modify those chars
  m_len = strlen(text);
  // m_text = text;  // WRONG! can't just copy the pointer,
                     // m_text then could have its string value changed elsewhere
                     // instead, m_text should hold its own char array
  m_text = new char[m_len+1];
    // can use a non constant array size when using new operator!
    // char built-in-type, so uninitialized
  // Blah* bp = new Blah[n];
    // on the other hand, default-constructors called n-times for the blah objects!
  strcpy(m_text, text);
}

// String::String()// make this as similar to the other constructor as possible
// {
//   // m_len = 0;
//   m_len = strlen("");
//   // m_text = new char[1];
//   m_text = new char[m_len+1];
//   // m_text[0] = '\0';
//   strcpy(m_text, "");
//   // now we can actually combine the constructors into one!
// }

String::String(const String& other)
{
  m_len = other.m_len;// yes, can access the private data members of any String
  // m_text = other.m_text; // no, has the same issue as compiler
                            // generated copy constructor, memory leaks
```

```
  m_text = new char[m_len+1];
  strcpy(m_text, other.m_text);

  // m_len = other.size();  // could alternatively copy using only member functions,
                            // but could be much less efficient!
  // m_text = new char[m_len+1];
  // for (int k = 0; k < m_len; k++)
  //   m_text[k] = other.charAt(k);
  // m_text[m_len] = '\0';
}

String::~String()
{
  // delete m_text; // BUT different forms of delete!
                    // for single dynamic objects, use delete p;
                    // for array of dynamic objects, use delete [] p;
  delete [] m_text;
}

// void String::assign(const String& rhs)    // we want to allow chaining operator
String& String::operator=(const String& rhs) // overloading assignment operator
                                             // can only overload existing operators
                                             // eg. s.operator[](k); or s[k];
{
  if (this != &rhs)
  {
    // delete [] m_text;// no more memory leak
    // m_len = rhs.m_len;
    // m_text = new char[m_len+1];
    // strcpy(m_text, rhs.m_text);

    String temp(rhs);// "copy-and-swap" approach for assignment operators
    swap(temp);      // works better with assignment,
                     // checks for self assignment, elegant!
                     // swap swaps the pointers
                     // temp is destructed at the end
  }
  return *this;      // allows for chaining assignment operator
}

void swap(String& other)
{
```

```
  char* temp = m_text;
  m_text = other.m_text;
  other.m_text = temp;

  int temp2 = m_len;
  m_len = other.m_len;
  other.m_len = temp2;
}

char* f(...); // returns a c-string

void h()
{
  String s("Hello"); // passing an array of characters to constructor
                     // double-quoted literals already have a zero byte appended
  String s2;         // acts same as if String s2("");
  String s3(f(...)); // what if null pointer? (can test)
                     // or no zero byte? (can't really deal with this problem)
  ...
  char line[1000];
  cin.getline(line, 1000);
  String t(line);
  cin.getline(line, 1000); // will overwrite line! since t simply holds a pointer,
                           // t would now hold a different String if m_text
                           // was initialized by pointer assignment
}

void g()
{
  for (...)
    h();
} // memory leak! dynamically allocated memory isn't released!
  // memory is a finite resource
  // "garbage" memory fills up over time unless there is a destructor
```

Default Parameter Values

- void f(int i, double d)
- void f(int i)
- if these functions act the same as if f(int i, 3.7), can combine into the **same** function using default values:
    - void f(int i, double d = 3.7)

        * this acts as both a one **and** two argument function
        * defaults to value of 3.7 for d if called with only one argument
    - eg. void g(int i, Point p = Point(), double d = 3.7)
        * acts as a one, two, and three argument function
        * can call:
            · g(10, q, 4.8);
            · g(10, q); acts as g(10, q, 3.7)
            · g(10); acts as g(10, Point(), 3.7)
        * **cannot** call:
            · g(10, , 4.8);
    - once you have provide a default value for an argument, **all** the other following arguments must also contain default values
    - default values can be **any** expression (even dynamically allocated objects), not necessarily constant
    - only **cannot** depend on the other parameters
- default values only go in the prototype (header), **NOT** the implementation
- can lead to ambiguities/compile issues
    - eg. function with two parameters (one default), and same name function with only one parameter
        * **cannot** call that function with one argument

Copy Constructors & Assignment Operator Overloading

```
// continued from String class above

void f(String t) // default copy constructor copies length and
                 // pointer to dynamic array from s to t
{
  // eg. t.set(0, 'J'); // here, value of s has changed out from under it!
                        // (because s and t both point to the same array)
  String u("WOW");
  ...
  // u = t;      // compiler generates an assignment operator that simply
                 // copies each data member from t into u
                 // in this case, leads to a memory leak:
                 // replaces and loses pointer to dynamic array ("WOW")
  // u.assign(t);// let's try writing a member function for assignment
  u = t;         // means u.operator=(t);
  ...
}                // t goes away at the end of a function,
                 // t's destructor is called, deletes dynamic array
```

```
void g()
{
  String s("Hello");
  f(s);
  ...
}                 // tries to delete the same dynamic array, undefined behavior
                  // as it is, can't pass our String class by value (copy)
                  // solution: we have to write our own copy constructor that
                  // creates a new dynamic array

// all the following examples involve copies and have the same issues
// tries to delete the same dynamic arrays twice
String s("Hello");
String t(s);

String h()
{
  String x;
  ...
  return x;
}

// Ex. Overloading copy constructor from String class

String::String(const String& other)
{
  m_len = other.m_len;
  m_text = new char[m_len+1];
  strcpy(m_text, other.m_text);

  // m_len = other.size();
  // m_text = new char[m_len+1];
  // for (int k = 0; k < m_len; k++)
  //   m_text[k] = other.charAt(k);
  // m_text[m_len] = '\0';
}

// Ex. Overloading assignment operator from String class

// void String::assign(const String& rhs)    // we want to allow chaining operator
String& String::operator=(const String& rhs)
{
```

```
  if (this != &rhs)
  {
    String temp(rhs);
    swap(temp);
  }
  return *this;
}
```

Copy Construction

- passing by value passes a **COPY**… how do copies work? copying is done by a *copy constructor*!
  - copy constructor creates an object based on another object
  - there are certain language constructs where copy constructor is called automatically (like default constructor)
    - eg. passing by value, returning an object, copy constructing (String t(s);)
      - alternate syntax for copy construction String s3 = s1;
    - initializing a brand new object
  - compiler writes its own **default** copy constructor, simply copies each member into copy (*shallow copy*)
  - using compiler generated copy constructor with **dynamically** allocated objects can lead to issues
    - error trying to delete dynamic objects **multiple** times
  - using copy constructor on classes composed of other classes with their own copy constructor
    - **automatically** calls those specific classes' copy constructors

Assignment Operator

- copying is **NOT** the same as assignment!
  - assignment operator does **NOT** call copy constructor
    - can only construct once
    - copy constructor theoretically wouldn't work in this case either, doesn't delete old dynamic memory
  - constructor: **nothing** in this to begin with (initialization)
  - assignment: there **is** stuff in this to begin with
  - compiler generated assignment operator copies each member (*shallow copy*)
  - make sure to check for errors from aliasing (don't sink your own ships)
- you can overload **operators** in c++
  - example syntax in prototype: void String::operator=(…)
    - eg. s.operator

- can only overload **existing** operators
- for assignment operator, convention is to return (*this) as a String reference in order to allow for **chain** assignment
  - assignment associates from right to left
  - eg. i = j = k = 0;
- have to ensure assigning an object to itself still works
  - eg. i = i;
  - otherwise, we may delete dynamic memory and then try to access there **again**

Inline Methods

- inline function code is **embedded** directly into the calling function by the compiler (for speed)
  - all functions with their body defined in their class are automatically inline
  - speed up program, but may make the .exe file larger

```
inline int bar(int a) // in function prototype
inline int foo::bar(int a)
inline ItemType foo<ItemType>::bar(ItemType a)
```

Data Structures and Algorithms

Linked Lists

- types of arrays so far:
  - *fixed size* array
    - number of elements **has** to be known at compile time
    - keep track of size
  - *dynamically allocated* array
    - now number of elements **doesn't** have to be known at compile time
    - but, now have to keep track of current size *as well as* max capacity
      - could be done with numbers or pointers
  - *resizeable* array (vector)
    - when going over capacity, dynamically allocate a **larger** array
      - how much to extend capacity by?
      - general rule of thumb, 1.5 * current capacity
    - copy all items into new array
    - delete old array, reassign to new array and increase capacity

- Pros and Cons of Arrays
  - **Pros**
    - same time to access each element
      - compiler knows where elements are stored contiguously
    - easy to add to the end of the array
  - **Cons**
    - hard to *maintain the order*, such as inserting in the **middle** or **beginning**
  - weigh pros and cons when considering data structures!
  - difficult to insert in the middle because memory for an array **is** contiguous
- how to make a data structure where the memory is incontiguous?
  - each value could hold a pointer to the next!
  - 65 -> 35 -> 69 -> 420 -> 666
  - how to signify end of the sequence? use nullptr
  - have a variable hold the first element of the list

Possible Approaches

- this is a **linked list**!
  - **first** element: **head** / **front**
  - each element is a **node**
  - **last** element: **tail** / **back**
  - to insert in the middle, instead of having to shift elements:
    - allocate a **new** node
    - put data of interest in **new** node
    - **new** node points to previous node's next node
    - previous node points to **new** node
  - only way to traverse right now is **forward** through the pointers, one step at a time
  - can't jump into the **middle** of the list
  - other considerations:
    - save pointer to the last node if doing operations often around the tail
      - but in a one item list, head = tail
    - have each node also point to the previous node (*doubly-linked list*)
      - much easier to insert and delete
      - don't have to go through the loop again to find the previous node
      - can also travel **backwards**
    - make a *circular list*
      - last node points back to the head
      - *no more nullptr's*, fewer special considerations (but head, tail,

empty **still** special cases)
        * add a *dummy node*
            · even though it doesn't hold a value, now there is **always** at least one node present
            · *no more special cases*!
            · acts as a placeholder
            · no longer needs a tail pointer

General Tips

- **Tips**:
    - general rule: any time you write p->something, *make sure to check*
        * p has previously been **given** a value
        * p is **not** the nullptr
    - **draw** diagrams!
        * check *typical situation* (activity in the middle), at the head, at the tail
        * emtpy list, one-element list
    - don't do things in the *wrong order*
        * general rule: set values in the new node **first**

Implementation

```
struct Node
{
  int   data;
  Node* next; // this is fine to the compiler, but can't have a node within a node
};

struct Node   // doubly-linked list
{
  int   data;
  Node* next; // successor
  Node* prev; // predecessor
}

Node* head;   // head points to the first element
              // if head == nullptr, list is empty

// Print out all values of the list

// while (head != nullptr)
// {
```

```
//   cout << head->data << endl;
//   head = head->next;
// }          // but now we can't go backwards, we lost head pointer!

for (Node* p = head; p != nullptr; p = p->next)
  cout << p->data << endl;

// Find an element of the list

Node* p;
for (p = head; p != nullptr && p->data != 42; p = p->next)
  ;
if (p == nullptr)
  cout << "Target value is not in the list" << endl;
else
  cout << "p->data is " << p->data << endl;
// check for nullptr everywhere! can't cout a nullptr
// what if the element isn't in the list?
// would this work with an empty list?
// short-circuit with p != nullptr!

// Insert element after another

Node* p;
for (p = head; p != nullptr && p->data != 42; p = p->next)
  ;
if (p != nullptr)
{
  Node* newGuy = new Node;
  newGuy->data = 32;
  newGuy->next = p->next;
  p->next = newGuy;
}
```

Stacks and Queues

- collection of data, but with a **limited** possibility of actions
- if so, implementing the data structure could be **simpler**

Stacks

- eg. the data structure runtime has to maintain:

    - calling a function suspends the current function
    - opens a new environment for the new function
    - falls back to the previous environment
    - all additions and removals happen at the **end** of the structure!
- new items **always** inserted at the end, only **last** item can be removed
- **LIFO** - *last in, first out*
- this is a **stack**!
- **Stack**:
    - create an empty stack
    - push an item onto a stack
    - pop an item from the stack (only when not empty)
    - look at the top item on the stack (only when not empty)
    - is the stack empty?
- more possible stack functionalities
    - how many items are in the stack?
    - look at any item in the stack

Queues

- what about similar, but with waiting in line? (first in line is served first)
- new items **always** inserted at the end, but only **first** item can be removed
- **FIFO** - *first in, first out*
- this is a **queue**!
- **Queue**:
    - create an empty queue
    - enqueue an item (at the tail / back)
    - dequeue an item (from the head / front) (only when not empty)
    - look at front item in the queue (no need to look at back of the queue) (only when not empty)
    - is the queue empty?
- more possible queue functionalities
    - how many items are in the queue?
    - look at the back item in the queue
    - look at any item in the queue
- these aren't low level data structures, both can be **implemented** with arrays or linked lists
- these have limitations, rather than *general purpose* arrays / linked lists

Priority Queues

- *priority queue* - extension of queue, every item has a *priority value*
    - items of same priority ordered by **FIFO** (so a queue)
- eg. printer printing pages: 1000, 1, 1

    - if treated as a normal queue, average wait time would be ~1000 for each print job
    - if instead we **prioritize** shorter print jobs, average wait time would be ~350 for each print job
        - should also include an "aging scheme" so that items in the queue for a long time eventually get processed
- unsorted sequence:
    - *insertion*: O(1) (just push end)
    - *remove*: O(N) (find highest priority item)
- sorted sequence:
    - *insertion*: O(N) (insert in order)
    - *remove*: O(1) (just pop end)
- BST:
    - *insertion*: O(logN)
    - *remove*: O(logN)
    - but if we're always approaching from the top side of the BST, high time complexity to rebalance the tree...
        - use a **heap**!

Syntax

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

int main()
{
  stack<int> s;
  s.push(10);
  s.push(20);
  int n = s.top(); // n == 20
  s.pop();         // void! (varies with different libraries / languages)
  if (!s.empty())
    cout << s.size();
}

#include <queue>
using namespace std;

int main()
{
  queue<int> q;
  q.push(10);         // not enqueue / dequeue
```

```
  q.push(20);
  int n = q.front(); // n == 10
  q.pop();
  if (!q.empty())
  {
    cout << q.size() << endl;
    cout << q.back() << endl; // c++ allows this
  }
}
```

Stack Implementations

- with an **array**:
  - have to maintain capacity and current size
  - to push, assign value and increment size
  - to pop, simply **decrement** the counter
  - thus top is the "end" of array
- with a **linked list**:
  - maintain just head pointer
  - to push, add node to the head, not the tail
  - thus treat the head as the top, **easiest** to get to
  - to pop, remove node from head

Queue Implementations

- with an **array**:
  - have to maintain capacity, head, and tail
  - head at position 0, tail at position one beyond last item in queue
    - number of items in queue then just tail - head
    - to enqueue, assign value and increment tail
    - if tail moved up to capacity
      - should we shift head and tail back to the beginning of the array?
      - could be array space usable before the head
      - eg. size of 2, but 98 spaces earlier in array with values that have been dequeued
      - instead, treat the array **circularly** and wrap back around!
      - called "*circular array*" or "*ring buffer*"
      - if head == tail, queue is either empty or full, how to distinguish?
      - instead maintain another variable for the current size
  - when dequeueing, would have to shift all values backwards
    - instead, simply *shift the head forward*!
    - head is position of the first item in queue

- with a **linked list**:
  - maintain head and tail pointers
  - to enqueue, add node to tail
  - to dequeue, remove node from head
  - unique case when empty or one element (*boundary case*)
  - usually no dummy node, singly linked (but tail still points to the end)

Evaluating Expressions with Algorithms

- eg. sample custom expression for a database could be:
  - dept = 'IT' and salary >= 70000 and name != 'SMITH'
- *Prefix Notation*:
  - does **not** need to consider precedence:
    * every operator takes a certain number of operands
  - f(x,y,z)
  - add(sub(8, div(6, 2)), 1)
  - + - 8 / 6 2 1
- *Infix Notation*:
  - **issues** with precedence and associativity
  - 8 - 6 / 2 + 1
- *Postfix Notation*:
  - does **not** need to consider precedence
  - 8 6 2 / - 1 +

Evaluating Postfix Expression

- How do we parse through an infix expression *in one go* to solve these expressions?
  - **translate** into postfix and do it!
  - eg. 8 6 2 / - 1 +
    * can use an *alogrithm with a stack* to solve
    * ever operator should match with two operands immediately before it
    * going from left to right:
      · if at an operand, **push** it onto stack
      · if at an operator, **pop** its operands, and **push** the result of the expression
      · should end with a *single result* in stack
    * 8 ~~6 2 /~~
    * ~~8 3 -~~
    * ~~5 1 +~~
    * 6

Translating Infix to Postfix

- How do we **translate** an infix expression into postfix?
  - also use an *algorithm with a stack*
  - going from left to right:
    - if at an operand, put it in new postfix expression
    - if at an operator and stack is empty, **push** it onto the stack
    - otherwise if precedence is **strictly** higher than top of stack **push** it onto the stack
      - if top of stack is open parenthesis, always **push**
    - otherwise if precedence is not **strictly** higher than top of stack, **pop** it from the stack, check again
    - if come to open parenthesis, **push** it onto stack
    - if come to close parenthesis, **pop** until we come to an open parenthesis
    - at end, **pop** everything off the stack
  - eg. 8 - ((2 + 1) * 2 - 3) + 4
  - **Expression**: 8 2 1 + 2 * 3 - - 4 +
  - **Operator Stack**: ~~- ((+) * -) +~~

Trees

- tree has a distinguished node (*root*), and one **unique** path from the root to **any** node
  - *parent* / *child* node relationships
  - *leaf* node has no children vs. *interior* node
  - *depth* of a node
  - *height* of the tree (max node *depth*)
- alternatively, every node *really only needs* two pointers / children (left and right)
  - called a **binary tree**
- every node has **data** and **pointers** to indicate structure of a tree
- operations with trees can be done more easily with *recursion*
  - *base cases* for trees:
    - empty tree, sometimes leaf nodes
  - *recursive cases* must solve smaller problem:
    - fewer nodes / breaking down into sub-trees
    - shorter height
- information can be passed up or down the tree recursively
- **processing patterns**:
  - *pre-order traversal*:
    - process node => process children of that node
  - *in-order traversal*:

        - for binary trees: process left subtree => process the node => process right subtree
    - *post-order traversal*:
        - process children of a node => process that node (passing info up)
    - *level-order traversal*:
        - process each level's nodes from left to right => process next level
    - all O(N)
    - draw counter-clockwise loop around tree to visualize:
        - pre-order: dot on left
        - in-order: dot under
        - post-order: dot on right

Example

```
struct Node
{
  string data;
  // Node* children[5];   // instead, variable number of children
  vector<Node*> children; // "singly-linked"
  Node* parent;           // "doubly-linked", optional
};

Node* root = nullptr;     // empty tree

// approach with a stack, or with recursion ...
int countNodes(const Node* t)
{
  if (t == nullptr)// only base case is empty tree, leaf node handled generally
    return 0;
  int total = 1;
  for (int k = 0; k < t->children.size(); k++)
    total += countNodes(t->children[k]);
  return total;
}

// print indented list
void printTree(const Node* t, int depth = 0); // default depth param in prototype

// void printTree(const Node* t) // have to keep track of depth for indents
void printTree(const Node* t, int depth)
{
  if (t != nullptr)
  {
```

```
    cout << string(2*depth, ' ') << t->data << endl; // temporary string
    for (int k = 0; k < t->children.size(); k++)
    {
      // cout << "  ";// won't work, subtrees don't know how deep they are
      // printTree(t->children[k], ++depth); // reassigning depth!
      printTree(t->children[k], depth+1);
    }
  }
}

// could overload printTree() with helper function
void printTree(const Node* t) { printTree(t, 0); }

int main()
{
  Node* windsor;
  ...
  printTree(windsor); // could overload printTree() or just use default parameter
}
```

Binary Trees

- any valid tree can be represented as a binary tree
- either empty, or a node with a left binary subtree and a right binary subtree
- how to *convert* a tree with arbitrary number of children to a binary tree?
  - left points to one of the children, right points to “siblings”

```
struct Node
{
  string data;
  Node* left;
  Node* right;
}

// eg. for a family tree
struct Node
{
  string data;
  Node* oldestChild;
  Node* nextYoungerSibling;
}
```

- *complete binary tree* is a binary tree that is filled at every level

  - except possibly the bottom, which is filled left to right
  - only one configuration for every number of filled nodes

Binary Search Tree

- special application of a binary tree
- either empty, or a node with a left BST and a right BST such that:
  - the value at every node in the left BST is <= the value at this node and
  - the value at every node in the right BST is >= the value at this node
- *traversals* always O(N), visit every item
- *search*: O(logN), similar to binary search on a sorted array
  - worst case O(N)
- example *insert* algorithm: (O(logN), search and then insert)
  - if empty, add as first node
  - otherwise, if less than current item, follow nodes to the left
  - otherwise, if greater than current item, follow nodes to the right
  - loop recursively
  - but this could lead to **unbalanced** sort trees with O(N) searching!
- example *removal* algorithm: (O(logN), search and then insert)
  - must keep track of parent
  - if target is a leaf node:
    - *unlink* from its parent (set parent to point to nullptr) and *delete* it
  - if has one child:
    - *relink* its parent to target's only child and *delete* it
  - if has two children:
    - **replace** its values with: (don't actually *delete* the node itself)
      - the largest value in its left subtree and *then delete* that node
      - ie. one step to the left and all the way to the right
      - the smallest value in its right subtree and *then delete* that node
      - ie. one step to the right and all the way to the left
  - thus, this utilizes a simpler deletion algorithm for the replacement
  - how do we choose which value to promote when deleting a node?
    - alternate left and right, or just randomly choose
- **balanced BST's**:
  - **AVL tree** - height of left subtree and right subtree differ by no more than one
    - *search*: always O(logN), always balanced
    - *insertion*: first insert using "dumb" algorithm and then rebalance the tree (O(logN), worse constant than "dumb" approach)
    - *deletion*: O(logN), worse constant than "dumb" approach
  - **2-3 Tree** - every node has two children (holds 1 value) or three children (holds 2 values)
    - all leaf nodes are at the same depth

    * *search*: don't have to go as deep
    * *insertion*: combination of splitting and promoting nodes
  - **Red-Black Tree** 2-3-4 trees - can represent structure using regular BST's
    * each node has extra boolean value defining its type / rules
    * no adjacent red nodes, every path from root to descendant has same number of black nodes
    * left and right subtrees differ up to a factor of 2
    * *less* work for balancing, but good enough
    * *all operations*: O(logN), better constant for insert / deletion than AVL tree
    * used with STL containers, eg. set

```
struct Node
{
  string data;
  Node* left;
  Node* right;
}

// inorder traversal
void printTree(const Node* t)
{
  if (t == nullptr)
    return;
  printTree(t->left);  // print left subtree
  cout << t->data << endl;
  printTree(t->right); // print right subtree
}
```

Hash Table

- we want to organize items indexed by ints, eg. 40,000 9-digit student IDs with student records
- if we had an array that is subscripted by the ID
  - lookup would be **constant** O(1), simply jump to the subscripted ID!
  - **BUT** wasted space in the array (1 billion slots in array, and what if the student record is large)
- now, have the array hold pointers instead of the actual student record
  - less space needed (only allocate 40,000 student records), but still empty slots in array
- now, organize the student ID's into various **buckets**
  - easy to determine what bucket student ID belongs to
  - can check even fewer elements in that bucket

        * buckets can be *empty* or have different *number* of items
    - eg. bucket number is last 4 digits of the student ID, 10,000 buckets with 4 elements each
- general *insert* algorithm:
    - determine which bucket
    - add item into bucket
        * order of items in a bucket doesn't matter, so few items, just use the simplest insert operation
        * eg. front of a linked list
- this is a **hash table**!
- **collision** - two different keys map to the same bucket
- **load factor** = # of items / # of buckets
    - tradeoff between many empty buckets and a long list
    - as load factor increases, collision chance and search time increase
        * 0.7 load factor
- hash table with **fixed** number of buckets is actually O(N) but with a really low constant of proportionality
    - eg. for the student ID's, $O(\frac{1}{10000}N)$
    - slower as buckets become more full
    - for large enough N, BST still wins out!
- *compared to BST*:
    - hash table has a *start-up* time
        * must initialize the buckets to nullptrs!
    - BST allows for easy sorted order *traversal*
        * possible solution with hash table:
            · every item in a bucket has a pointer to the next ordered item
            · probably doubly-linked
    - so, go with hash table **unless** we want to visit items in order
        * but if we don't have to visit sorted items often...
            · simply add all items from hash table into a vector, then sort the pointers / values
            · this is O(NlogN), worse than O(N) for BST traversal
            · but BST costs O(logN) for *every* insert
        * paying O(logN) cost somewhere, hash table may be more efficient

Dynamic / Resizable Hash Table

- *fixed* limit on the load factor
- resize hash table and **reallocate** entire table with new buckets once limit is reached
    - eg. twice as many buckets
    - but now keys may be rehashed to different buckets (mod buckets gives a different value)

        * must be re-added
    - then add the new item
    - newly reallocated table has low load factor, then starts to fill up until max load faactor
- has constant time O(1), but every so often has to do a rehash O(N) when inserting
    - but **on average**, still O(1)! (*amortized*, averaged)
    - however, big rehashes can cause spikes of bad time complexity
- instead, use **incremental** rehashing:
    - rehash some of the table at one time instead of entire table (eg. 5 elements)
    - when new table is partially rehashed:
        * search, insertion, and deletion must check *both* tables (still O(1))
            · every insertion / deletion will incrementally rehash more elements from the first table
    - but now, **every** insertion has a constant runtime!
        * ie. 20% of the time, in two-table mode, have a higher constant runtime

Hash Functions

- **hash function** - key -> integer (then take integer and scale to number of buckets)
    - then, integer mod number of buckets to scale to buckets
    - eg. strings, add up the ACII codes; /1000 or %1000 for numbers
        * could also multiply ASCII codes
        * selecting digits, folding (adding digits), modulo arithmetic, string to int
    - use a prime number for number of buckets
        * eg. h(x) = x % numBuckets
        * reduces collisions
        * deals with distributions that aren't uniform
    - want cheap, deterministic, uniformly distributed result
- collision resolution schemes
    - *linear probing* - **closed** hash - data stored in a fixed-size array
        * "closed" number of buckets
        * if bucket is occupied, scan down from bucket until we hit the first open bucket
            · buckets represented as arrays
            · if at end of buckets, wrap around to the front
            · when searching, now have to probe linearly until we find our value or hit empty bucket
        * issue of primary clustering (values are as close to intended bucket

as possible)
        - longer probing sequences and time to seach
        - difficult to delete items, limit on # of items to hold!
        - deleting items may disrupt the linear probing
      * less efficient, faster growth in search time
      * more memory efficient
      * average number of checks for insert/find: $1/2(1 + 1/(1 - L))$
  - *separate chaining* - **open** hash tables - data stored in linked-lists, **not** size-limited
      * in case of collision, just add another value to the linked-list
      * more efficient, slower growth in search time
      * average number of checks for insert/find: $1 + L/2$
- sample hash function for strings:

```
// FNV-1 variation (Folwer-No-Vo) -
// fast while maintaining low collision with high dispersion
unsigned int h = 2166136261U; // ensure positive, U removes compiler warning
                              // for large integer, ie. "offset basis"
for (int k = 0; k < key.size(); k++)
{
  h += key[k];
  h *= 16777619; // FNV_prime, eg. 2^40 + 2^8 + 0xb3
}

// STL hash function
#include <functional>

unsigned int hashFunct(const std::string& key)
{
  std::hash<std::string> str_hash; // define string hashing object
  unsigned int hashValue = str_hash(key);
  return hashValue % NUM_BUCKETS;
}
```

Heap

- **(max) heap** - a complete binary tree where each node has a value >= all the nodes in its subtrees
  - order and what's in each child isn't important...
- eg:

```
    90
  60    80
```

```
  40  50  70  20
10  30
```

- **min heap** - a complete binary tree where each node has a value less than or equal to all the nodes in its subtrees
- is a complete binary tree
- *insertion*:
  - first add the item to make a complete binary tree (only one possible location)
  - then, if the item is greater than its parent, simply swap them
    - "**bubble**" up the new items to its proper position (reheapify)
    - O(logN) to bubble up
- *removal*: (only want to remove max item)
  - remove root
  - promote bottom-most item (of complete binary tree) and **promote** it to the root
    - ie. remake tree into a complete binary tree
  - "**trickle**" down the root to its proper position (reheapify)
    - at each level, make sure parent is larger than both its children
  - O(logN) to trickle down
- **heap sort / partial heap sort**:
  - insert all items into a heap (O(NlogN)) and then extract however many items (OKlogN)
  - uses heaps to sort an array partially, eg. top N items
  - approximately O(NlogN)
- but in either case, have to calculate position of the bottom-right node / parent and children nodes in the tree
  - there is another representation that allows for this calculation to be done in constant time
    - use **arrays**!
- number nodes from left to right, top to bottom from 0 to N-1, and use that number as the index in an array
  - given node number i, is there a formula that generates the parent and children of i?
    - (i-1)/2 gives the parent of i (using integer division)
    - 2i+1 and 2i+2 gives the children of i, provided they are less than the number of nodes
  - now it is constant time to find the bottom-most item of the tree! (given at N-1)
  - can still use the same bubble up / trickle down algorithms (they still have the same complexity)
  - use array *operations* but visualize with a tree!

Tables

- made up of multiple *records* (represented by a struct)
  - each record has related *fields*
- a table can be represented by a collection of structs
- how to efficiently search by different fields (eg. name, ID, phone number)?
  - retain one table, but:
  - use secondary data structures called *indexes*
    - eg. maps mapping name to slot in vector, id to slot in vector, etc.
  - when updating and deleting records, must update across all indexes
- use BST to store data in order
- use hash table for fast searching when order isn't important (eg. phone numbers)

Graphs

- made up of **nodes** (vertices) and **edges** (arcs) connecting them
  - graphs can have *undirected* or *directed* edges
    - *cyclic* graph where a path leads back to a node vs. *acyclic* graph
    - directed, acyclic graphs are relatively common
- graphs can model many different types of problems
- there are many graph algorithms of different time complexities
  - "graph theory"
  - eg. *topological sort*
- *implementation*:
  - use a 2-d array if lots of edges between vertices but few vertices
  - each element indicates if there is an edge between vertex i & j
    - called an **adjacency matrix**
  - multiplying an adjacency matrix by itself yields a matrix showing us vertices 2, 3, etc. edges apart
  - could also use an array of lists if few edges and many vertices
    - eg. list graph[n]
- breadth-first and depth-first traversals also apply to graphs
  - forward until dead end vs. exploring graph in growing concentric circles
- can have *weighted* edges on graphs
  - use Dijktra's Algorithm to find the shortest path between any two nodes in a graph
    - uses settled / unsettled vertices

Standard Template Library (STL)

- we've seen stacks and queues in the STL
- using STL requires *using namespace std*

Vectors

- dynamically resizable array
- error checking **not** built into subscript operator, because it would be expensive!
  - to be safe, check if subscript is out of bounds
- at() function *does* do bounds checking
- inserting in the middle is expensive
- jumping to different elements is cheap (via brackets)

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

vector<int> vi;
vi.empty(); // true
vi.push_back(10);
vi.push_back(20);
vi.push_back(30);   // no push_front()
cout << vi.size();  // writes 3 (doesn't work for arrays!)
cout << vi.front(); // writes 10
cout << vi.back();  // writes 30
vi[1] = 40;         // gives a reference to that value, vi[3] = 50;
                    // would be undefined behavior
for (size_t k = 0; k < vi.size(); k++) // size_t, unsigned integer type
  cout << vi[k] << endl;
  // writes 10 40 30, one per line
vi.pop_back();      // undefined if vector is empty, vi.empty()
for (size_t k = 0; k < vi.size(); k++)
  cout << vi[k] << endl;
  // writes 10 40, one per line
vi.at(1) = 60;
vi.at(3) = 70;      // throws exception

vector<double> vd(10);          // vd.size() is 10, each element is 0.0
vector<string> vs(10, "Hello"); // vs.size() is 10, each element is "Hello"
int a[5] = { 10, 20, 30, 40, 50 };
vector<int> vx(a, a+5);         // passing a range of values in a container
// vx.size() is 5, vx[0] is 10, vx[1] is 20, ..., vx[4] is 50

// optionally:
```

```
vector<int> vx = {10,20,30,40,50}; // C++11
vector<int> vx {10,20,30,40,50};   // C++11
vector<int> vx(100);// all 100 elements are default initialized / constructed

// common mistake:
vector<int> vi;
// vi[0] = 10; // undefined! vi.size() is 0, no elements!
```

Implementation

- *data members*:
  - dynamically allocated array
  - size
  - capacity
- empty:
  - nullptr to array
  - size, cap of 0
- at *full* capacity:
  - allocates a larger capacity, copies values, deletes old values, retargets pointer, update capacity
- this means saving a pointer can leave a dangling pointer *later on* after a push-back!
  - *anything pointing* to the vector can become **invalidated**!
  - remember the position instead of saving a pointer...

List

- linked list
  - similar functions as the vector class (push_back, pop_back, front, back, size, empty)
    * but *also* has push_front and pop_front
- subscript and at() are **not** given with a list!
  - very expensive
- inserting / deleting in the middle is not as expensive
- jumping to elements is expensive

```
#include <list>
using namespace std;
list<int> li;
li.push_back(20);
li.push_back(30);
li.push_front(10);
cout << li.size();  // writes 3
```

```
cout << li.front(); // writes 10
cout << li.back();  // writes 30
li.push_front(40);
li.pop_front();     // also pop_back()
```

Set

- *associative containers*:
  - designed so that searching for items is efficient
- eg. a set is a collection that **doesn't** allow duplicates
  - more efficient than searching in an unordered container
  - no [] operator!
    - but *can* iterate over the container
      - guaranteed to iterate through set *in order* using `<` operator
      - requirement for items in set to have `<` operator *defined*
      - set's definition of duplicate:
      - !(a `<` b) and !(a `<` c), then a == b
      - this definition only works well with total ordering (eg. not case-sensitive strings)
      - other containers use different operators for ordering
- set and multiset use a Red-Black Tree (BST) for implementation
  - standard requires logarithmic time for all operations
  - multiset allows for duplicate values
- unordered_set / unordered_multiset use a Hash Table implementation
  - #include
  - unordered_set.find(x)

```
#include <set> // defines std::set and std::multiset

set<int> s;
s.insert(10);
s.insert(30);
s.insert(20);
s.insert(10);
cout << s.size(); // writes 3
if (s.find(20) == s.end())
  ... not found ...
for (set<int>::iterator p = s.begin(); p != s.end(); p++)
  cout << *p << endl;
// guaranteed to write out 10 20 30
s.erase(10);
```

Map

- another *associative container*
- maps one related value to another
  - eg. people to their phone number, ie. strings to ints
- maps can only associate in **one** direction!
  - to search efficiently in both directions, need *two* maps
- **doesn't** allow duplicates
- map class stores each association in a **struct** variable:
  - eg. string first, int second
- use iterators to traverse through
  - can access first and second variables similar to a struct
- maps are automatically maintained in *alphabetical* order for the key!
  - thus if associating a complex data type, operator `<` **must** be defined to order items
    - but only for the **left-hand** side of the map (key)
- map and multimap use a Red-Black Tree (BST) for implementation
  - O(logN) for all operations
- multimap allows for duplicate keys
- unordered_map / unordered_multimap use a Hash Table implementation
  - requires `<` , == operator
  - a hash function for the type
    - provided for built-in type, strings, thread-ID's, etc.
    - have to write hash function for other custom types

```cpp
#include <map> // defines std::map and std::multimap
using namespace std;

map<string, double> ious;
string name;
double amt;
while (cin >> name >> amt)
  ious[name] += amt; // overloaded subscript operator,
                     // can add new maps or update maps in this way

cout << ious.size() << " people owe me money" << endl;
for (map<string, double>::iterator p = ious.begin(); p != ious.end(); p++)
  cout << p->first << " owes me $" << p->second << endl;

map<string, int> name2Phone;       // string to int, but not the other way around!
name2Phone["Carey"] = 0001112222;
name2Phone["Joe"] = 2223334444;
// name2Phone[1112223333] = "Ed"; // doesn't compile
```

```
map<string, int>::iterator it;
it = name2Phone.find("Joe");      // locate an association
if (it == name2Phone.end())
  cout << "not found!" << endl;
else
{
  cout << it->first; // look at the pair of values pointed to by iterator
  cout << it->second;
}

map<int, string> fones2Names;      // int to string

// dealing with duplicate keys:

// use a multimap...
// equal_range() returns a pair of iterators indicating the range of equal keys
multimap<string, double> borrowings;
pair<multimap<string, double>::iterator, multimap<string, double>::iterator> pr =
                                                borrowings.equal_range("fred");
if (pr.first == pr.second)
  ...not found...
else
  for (multimap<string, double>::iterator p = pr.first; p != pr.second; p++)
    cout << "fred borrowed $" << p->second << endl;

// alternatively, use a map with another STL container for more than one value
map<string, list<double>> borrowings;

// unordered_map as a hash table
#include <unordered_map>
using namespace std;

unordered_map<string, double> ious;
```

Auto Keyword

- how to *reduce* verbeage when declaring iterators?
- declaration: sometype v = initializer;
- **auto** keyword sets variable type to the initializer automatically in a declaration
  - type is determined at compile time (NOT a varying type)

```
// pair<multimap<string,double>::iterator, multimap<string,double>::iterator> pr
//  =  borrowings.equal_range("fred");
auto pr = borrowings.equal_range("fred");
  // returns a pair, pr will be a variable of type pair, less redundant

// for (multimap<string,double>::iterator p = pr.first; p != pr.second; p++)
for (auto p = pr.first; p != pr.second; p++)

// map<string,int>::iterator p = ious.find("fred");
auto p = ious.find("fred");

auto* p = new Circle(...); // p is Circle*
auto* p = f(...);          // if f returns a Shape*, p is Shape*
```

Iterators

- nodes and pointers in the list class are hidden from the user (they are abstracted away)
- instead provide something that acts like a pointer, called an **iterator**
  - no longer have to do messy pointer work
  - abstract way of traversing through elements
  - iterator is a class type, begin() and end() return iterators
- we can follow iterators with dereference operator, and use ++ and –
  - but can **not** use general pointer arithmetic, eg. adding 7 to an iterator
    - but you **can** with *VECTORS*! (cheaper!)
- using iterators, possible to:
  - *loop* / *traverse* through a list
  - *insert* a value (value is inserted just *before* the passed iterator)
    - returns an iterator pointing to the inserted value
    - passed iterator remains the same (*unless* used with a vector, then becomes **UNDEFINED** to use!)
      - when inserting a value into a vector, the vector may have been **RESIZED** and **REALLOCATED**!
      - when reallocation occurs, **all** iterators become invalidated!
      - but when begin() and end() are *recalled* they return a valid iterator
  - *erase* a value (erase value passed iterator is pointing to)
    - returns an iterator pointing to the value after the erased value
    - the passed iterator becomes **UNDEFINED** to use!
- iterators *can't* hold a value of nullptr!
- sometimes need to use a const iterator (when a container is passed as const reference)

- notes on lists: (doubly-linked list)
  - *cannot* use [] operator
  - *cannot* use comparison operators with iterators (`>, <, <=, >=`), there is no valid ordering
  - using iterator to traverse a list is always safe
- notes on vectors: (dynamic array)
  - *can* use [] to access elements in a vector
  - comparison operators are *valid*
  - but could **invalidate** iterators when using push_back()

```
structure<data type>::iterator it       // a pointer to an element in a container
structure<data type>::const_iterator it // constant iterator for constant reference
li.begin(); // an iterator pointing to beginning of the list
li.end();   // an iterator pointing to just after last value of the list
            // can't follow the iterator here, must decrement first!
li.back();  // an iterator pointing to the last element of the list

// can use these iterators to initialize a vector
list<double> ld(10);                    // ld.size() is 10, each element is 0.0
list<string> ls(10, "Hello");           // ls.size() is 10, each element is "Hello"
vector<string> vs(ls.begin(), ls.end()); // vs.size() is 10, vs[0] is "Hello",
                                         // vs[1] is "Hello", ..., vs[9] is "Hello"

// Iterators with Lists:

for (list<int>::iterator p = li.begin(); p != li.end(); p++)
  cout << *p << endl;  // writes 10 20 30, one per line
// can also use rbegin() and rend() for reverse traversal
// (still would increment to advance the iterator)
// can also use constant iterators to promise not to change the list

list<int>::iterator p = li.end();
p--;
p--; // given a list with {10, 20, 30}, p points to 20
// p -= 2 won't compile

list<int>::iterator q = li.insert(p, 40); // insert() inserts just before iterator
// list is now {10, 40, 20, 30}
// q points to 40 (the inserted value), p still points to 20

list<int>::iterator q = li.erase(p);// erase() erases value iterator points to
// list is now {10, 40, 30}
// q points to 30 (value after erased value), now UNDEFINED to use p!
```

```
for (list<int>::iterator p = l.begin(); p != l.end();)
  if (*p == 30)
    p = l.erase(p); // remove value pointed by p, and reassign the
                    // return value back to it (no need to increment)
  else
    p++;

// Iterators with Vectors:

vector<int>::iterator p = vi.end() - 2;
// given a vector with {10, 20, 30}, p points to 20

vector<int>::iterator q = vi.insert(p, 40);
// vector is now {10, 40, 20, 30}
// q points to 40 (the inserted value), now UNDEFINED to use p!
// (vector could have been RESIZED and values REALLOCATED!)

p = vi.erase(q);
// vector is now {10, 20, 30}
// p points to 20 (value after erased value), now UNDEFINED to use q!
```

Iterators with Templates

A find function:

```
// b points to beginning, e points to just after the end
int* find(int* b, int* e, const int& target)
{
  for ( ; b != e; b++)
    if (*b == target)
      break;
  return b; // going to be generalized later for iterators, can't return nullptr
}           // we return just past the end value if target not found

// Template Version:
template<typename T>
T* find(T* b, T* e, const T& target)
{
  for ( ; b != e; b++)
    if (*b == target) // T must have an equality comparison operator
      break;          // but this WOULDN'T work when called with "Fred"
                      // (a pointer to a char array!)
```

```
  return b;// we want to decouple the target type from the traversing type
           // (to use with pointers / iterators)
}
string* sp = find(sa, sa+4, string(“Fred”));
  // would need a temporary string object to use correct equality comparison

// Template Version w/ Iterators:
template<typename Iter, typename T>
Iter find(Iter b, Iter e, const T& target)
{
  for ( ; b != e; b++)
    if (*b == target) // CAN compare string (*b) to a char pointer!
      break;
  return b;
}

int* p = find(a, a+5, k);
if (p == a+5)
  ... not found ...

list<string>::iterator q = find(ls.begin(), ls.end(), "Fred");
if (q == ls.end())
  ... not found ...

vector<int>::iterator r = find(vi.begin(), vi.begin()+5, 42);
if (r == vi.begin()+5)
  ... not found ...
```

STL Algorithms

- #include
- find(v.begin(), v.end(), val)
    - also works with arrays: find(&a[0], &a[5], 19)
- reverse(v.begin(), v.end())
- sort(v.begin(), v. end()) (uses quick sort, not supported with list, needs random access)
    - list has its *own* sort member function (special case, uses merge sort)
- set_intersection() - compute intersection of two sorted collections of data
- can pass a **predicate function** with many of these functions:
    - calls that predicate function on each of the elements!

```
template<typename Iter, typename Func>
Iter find_if(Iter b, Iter e, Func f)
{
  for ( ; b != e; b++)
    if (f(*b)) break; return b;
}

bool isNegative(int k) { return k < 0; }
bool isEmpty(string s) { return s.empty(); }

int main()
{
  vector<int> vi;
  vector<int>::iterator p = find_if(vi.begin(), vi.end(), isNegative);
  if (p == vi.end())
    ... not found ...
  list<string> ls;
  list<string>::iterator q = find_if(ls.begin(), ls.end(), isEmpty);
}

bool isGreater(int i, int j)
  { return i > j; }
bool makesLessThan(const Employee& e1, const Employee& e2)
  { return e1.salary() < e2.salary(); }
bool hasBetterRecord(const Team& t1, const Team& t2)
{
  if (t1.wins() > t2.wins())
    return true;
  if (t1.wins() < t2.wins())
    return false; return
  t1.ties() > t2.ties();
}

int main()
{
  vector<int> vi;
  sort(vi.begin(), vi.end()); // uses <
  sort(vi.begin(), vi.end(), isGreater);
  Employee ea[100];
  sort(ea, ea+100, makesLessThan);
  vector<Team> league;
  sort(league.begin(), league.end(), hasBetterRecord);
```

```
  list<int> li;
  li.sort(); // uses <
  list<Employee> le;
  le.sort(makesLessThan);
}

// pointers to functions:
int (*ptr) (int); // ptr points to any function that
                  // returns an int and takes a single int
ptr = squared;
cout << ptr(5);
```

Recursion

- to solve larger problems, *break down* into smaller problems
  - but there needs to be one or more *base cases* (stopping condition)
    - ie. situations that can be solved without a recursive call
      - proof of termination
  - *recursive cases* - must solve a smaller problem, closer to a base case (simplifying step)
    - usually:
      1. divide input problem in **half** (merge sort)
      2. operate on input that is *one smaller*
      3. eg. back to front (last and the rest), front to back (first and the rest), divide and conquer
- eg. to **sort** an unsorted pile of N items
  - if (N > 1)
    - split into two unsorted piles of N/2 items
    - sort left subpiles
    - sort right subpiles
    - **merge** two resulting sorted subpiles into one sorted pile
  - base cases: N is 1 or 0
- some issues
  - odd N? merge algorithm still works
  - down to one item?
    - would **continue** trying to sort/split a single item
    - **infinite** recursion!
- some steps:
  1. write function header
  - find out what / num of arguments, return type

2. define "**magic**" function
   - solves problem, but only with a smaller subset / n
   - **same** parameters and return type!
3. add base case code
   - deal with **simplest** possible inputs
4. solve problem w/ magic function (*recursive leap of faith*)
5. remove magic...
6. validate function
   - test with simplest possible inputs
   - test with incrementally more complex inputs

- recursion **vs.** iterative solutions?
  - for simple recursion (one recursive call), easily implementable with iteration
    * iterative solution is usually *faster*
  - for more than one recursive calls, *simpler* to write and implement
    * divide and conquer recursion
- recursion can be **expensive**! (lots of local variables)
  - **DON'T** use recursion when it isn't neccesary!
  - don't let recursive calls get too deep
- can use *recursive helper functions* to simplify complex parameters
  - eg. binary search's confusing bot / top parameters

Recursion Examples

Sorting Algorithm

- **indexing**: a subarray starts at b and goes to one item before end

```
void sort(int a[], int b, int e)
{
  if (e - b > 1) // more than one element
  {
    int mid = (b + e) / 2;
    sort(a, b, mid);
    sort(a, mid, e);
    merge(a, b, m, e);
  }
}
```

Finding a Target

```
bool contains(int a[], int n, int target)
{
```

```
  if (n <= 0)
    return false;
  if (a[0] == target)
    return true;
  // pointer airthmetic, start at second element of array
  return contains(a+1, n-1, contains);
}
// Order matters!
bool contains(int a[], int n, int target)
{
  if (n <= 0)
    return false;
  // calls n recursive calls even if early element is already target
  if (contains(a+1, n-1, target))
    return true;
  return a[0] == target;
}
// Back to front!
bool contains(int a[], int n, int target)
{
  if (n <= 0)
    return false;
  if (a[n-1] == target)
    return true;
  return contains(a, n-1, contains);
}
```

- some initial issues:
  - never return false
  - have to check subscript / pointer arithmetic is valid
  - so, return false if n `<=` 0
- optimize to reduce recursive calls
- front to back vs. back to front approach

Solving a Maze

```
bool solve(start, goal)
{
  if (start == goal) // if we're at the goal
    return true
  mark this position as visited // check where we've been
  for each direction
```

```
    if moving one step is possible and not been visited,
      if (solve(pos reached by that move), goal)
        return true
  return false
}
```

- some initial issues:
  - don't check where we've been
  - starting at the goal
- is recursion solving a simpler problem?
  - distance not necessarily shorter, because of *walls*
  - mark where we have visited, reducing unvisited places

Big-O

- how do we categorize the speed of an algorithm?
  - can't just compare runtimes
    - must account for hardware differences
  - count steps in the algorithm in terms of N
    - any basic operation is a step
    - but we don't really care for speed with a small N
    - or about the exact details
  - so we can disregard the lower-order terms
    - can also generalize and disregard the leading coefficient
      - finding the worst case scenario
- a function $f(N)$ is $O(g(N))$, if there exists N_0 and k s.t. for all $N >= N_0$, $f(N)` <= ` k * g(N)$
- f(N) is "order g(N)"

Operations with Arrays

- O(1) - accessing elements in an array (*constant time*)
- O(logN) - binary searching elements in an array (*logarithmic time*)
  - always dividing in half, $\log_2$N gives number of comparisons
    - can disregard base 2, all logarithms are proportional
  - grows very slowly over constant time
- O(N) - searching elements in an array (*linear time*)
- O(NlogN) - grows very slowly over linear time (*linear logarithmic time*)
- O(2^N) - printing all possible subsets in a collection
  - one more item doubles the time

More Examples

Process for Evaulating Big-O

- work *inside-out*, starting from the most deeply nested statement
- have to account for *hidden* loops
  - eg. loops in embedded functions

```
for (int i = 0; i < N; i++)
  c[i] = a[i] + b[i];
// basic operations all take a constant amount of time
// array subscript operator is simply multiplication and addition
// every loop takes constant time, runs N times, this algorithm is O(N)

for (int i = 0; i < N; i++)   // entire algorithm is O(N^2)
{                             // body of loop is O(N)
  a[i] *= 2;                  // O(1)
  for (int j = 0; j < N; j++) // happens N times, O(N)
    d[i][j] = a[i] * b[j];    // array subscripts are O(1)
}
// work inside out, from most deeply nested statement
// shouldn't assume every loop in a loop is O(N^2)...
// if inner loop only ran up to 100, algorithm would be O(N)
// little trivial differences, eg. starting at 1 or going to N-1,  aren't important

for (int i = 0; i < N; i++ )     // entire algorithm is O(N^2)
{                                // body of loop is O(N)
  if (find(a, a+N, 10*i) != a+N) // find() is O(N), comparison is O(1)
    count++;                     // O(1)
}
// not just O(N) simply because there is one loop! (find() itself is O(N)!)
// have to count the hidden for loops!
// how do we account for if statements?
// for worst case scenario, assume body of if will always execute

for (int i = 0; i < N; i++)   // O(N^2)
{                             // O(i) + O(1) = O(i)
  a[i] *= 2;                  // O(1)
  for (int j = 0; j < i; j++) // O(i)
    d[i][j] = a[i] * b[j];    // O(1)
}
// sum of all numbers from 1 to N-1 is 1/2 N^2 - 1/2 N
// still O(N^2)! (ignore constant proportionality, for now)
```

```
for (int i = 0; i < N; i++)   // O(N^2logN)
{                             // O(NlogN)
  a[i] *= 2;                  // O(1)
  for (int j = 0; j < N; j++) // O(NlogN)
    d[i][j] = f(a, N);        // O(logN)
}`
// given f() is O(logN)

// example with STL set
void addItems(set<int> &s, int q)
{
  for (int i = 0; i < q*q; i++)   // O(Q^2logQ^2)
    s.insert(i);                  // O(logN) to insert
}                                 // in worse case, N will eventually become Q^2
```

Multiple Variables

- what if the Big-O depends on multiple parameters / variables?
  - have to keep track of both parameters!
    - either variable could dominate the other

```
for (...R times...)   // O(RClogC)
  for (...C times...) // O(ClogC)
    f(...c)           // O(logC)
```

STL Cheat Sheet

- Big-O of STL container operations

Table 1: STL

Container	Insertion	Deletion	Access	Find
list (linked list)	O(1)*	O(1)*	O(1) - top or end, O(N) - middle	O(N)
vector (resizeable array)	O(N) - top or middle, O(1) - end	same as inserting	O(1)	O(N)
set (set of unique items)	O(logN)	O(logN)	NA	O(logN) - tree
map (maps one item to another)	O(logN)	O(logN)	O(logN)	O(logN) - tree

Container	Insertion	Deletion	Access	Find
queue and stack	O(1) - push	O(1) - pop	O(1) - top	NA

*to get to the middle, may first have to iterate through N items at O(N)

Sorting Algorithms

General rules:

1. don't choose a sort until you know the requirements of the problem
2. always choose the simplest sorting algorithm possible that meets your needs

stable sort:

- takes into account initial ordering when sorting
- maintains the order of similar-values items

unstable sort:

- re-orders items without taking into account their initial ordering

Selection Sort

- loop through unchecked indices, find **minimum** and swap
  - even if no swap is necessary (ie. minimum at current index)
    - this algorithm would still check the rest of the items
- no best or worse case, *always* O(N^2)
- learning nothing on each pass
  - only useful when swapping/movements is really *expensive* compared to comparisons
- example sort:

```
|5 3  7  6  1  8  2  4
1 |3  7  6  5  8  2  4
1  2 |7  6  5  8  3  4
1  2  3 |6  5  8  7  4
1  2  3  4 |5  8  7  6
1  2  3  4  5 |6  7  8
1  2  3  4  5  6 |7  8
1  2  3  4  5  6  7 |8
```

- Implementation:

```
void selectionSort(int A[], int n)
{
  for (int i = 0; i < n; i++)
  {
    int minIndex = i;
    for (int j = i+1; j < n; j++)
    {
      if (A[j] < A[minIndex])
        minIndex = j;
    }
    swap(A[i], A[minIndex]);
  }
}
```

Bubble Sort

- check **pairs** of items, put them in sorted order
- after one loop, the largest item is *guaranteed* to be shifted to the last item
- loop again, stopping right before that largest item
  - we can skip steps by setting the stopping position at the last swap
- *best case*: O(N) - already sorted
- *worst case*: O(N^2) - reverse order, swapping all the time
- *average case*: still O(N^2) - but constant of proportionality is *half* as bad
- example sort:

```
5  3  7  6  1  8  2  4|
3  5  6  1  7  2  4| 8 # 8 is largest item
3  5  1  6  2  4| 7  8
3  1  5  2  4| 6  7  8
1  3  2  4| 5  6  7  8
1  2| 3  4  5  6  7  8 # put stopping position at last swap
|1 2  3  4  5  6  7  8 # if no swaps, array is sorted
```

- Implementation:

```
void bubbleSort(int Arr[], int n)
{
  bool atLeastOneSwap;
  do
  {
    atLeastOneSwap = false;
    for (int j = 0; j < (n-1); j++)
    {
```

```
      if (Arr[j] > Arr[j + 1])
      {
        swap(Arr[j],Arr[j+1]);
        atLeastOneSwap = true;
      }
    }
  }
  while (atLeastOneSwap == true);
}
```

Shell Sort

- built on underlying sort called *h-sorting*
  - pick a value for h
  - for every element in the array:
    - if a[i] and a[i+h] are out of order, swap them
  - if any elements were swapped in the last pass, repeat again with same h
  - eg. when an element is 3-sorted, every element is smaller than the element 3 items later
  - 1-sorting *is* a bubble sort
- *shell sort* approach:
  - select a sequence of decreasing h-values **ending** with 1
    - eg. 8, 4, 2, 1
  - then 8-sort, 4-sort, 2-sort, finally 1-sort the array
- ~$O(N^{1.25})$ depending on the decrement sequence!
  - does not require much extra storage

Insertion Sort

- similar to ordering a hand of cards as they are dealt
  - insert cards picked up in order into the hand
    - examine cards in hand one by one to figure out where to put the next card
  - hand is always **in order**
- *best case*: O(N) - already sorted
- *worst case*: $O(N^2)$ - reverse order, have to compare with all previous items
- *average case*: $O(N^2)$ - but constant of proportionality is *half* as bad as previous algorithms
- *slightly* better than bubble sort
- **except** if all items are *no more* than some **constant** distance out of place, only shifting constant times, then insertion sort is O(N)!

- example sort:

```
5 |3  7  6  1  8  2  4
3  5 |7  6  1  8  2  4
3  5  7 |6  1  8  2  4 # got lucky, only one comparison
3  5  6  7 |1  8  2  4
1  3  5  6  7 |8  2  4 # lots of comparisons and shifting
1  2  3  5  6  7  8 |4
1  2  3  4  5  6  7  8|
```

- Implementation:

```
void insertionSort(int A[], int n)
{
  for(int s = 2; s ≤ n; s++)
  {
    int sortMe = A[ s - 1 ];
    int i = s - 2;
    while (i ≥ 0 && sortMe < A[i])
    {
      A[i+1] = A[i];
      --i;
    }
    A[i+1] = sortMe;
  }
}
```

Merge Sort

- **merging** two sorted piles is O(N)
- but what is the runtime for a recursive algorithm?
- $T(N) = 2T(N/2) + O(N)$ (*recurrence relation*) ~ O(NlogN) (*all cases*)
  - much better runtime than O(N^2)
    - **except** if all itmes are *no more* than some constant items out of place
      - may take longer than insertion sort
- doesn't require *random access* (fine with arrays, vectors, lists)
  - list's sort member function uses merge sort
    - lists are easy to swap / copy items, so could be *faster* than quick sort if expensive to move items
- because merge function needs secondary arrays to merge, this can slow things down compared to quicksort
- requires extra storage (O(N)) to have a good constant of proportionality
- Merge() Implementation:

```
void merge(int data[],int n1, int n2)
{
  int i=0, j=0, k=0;
  int* temp = new int[n1+n2]; // needs a temporary array!
  int* sechalf = data + n1;

  while (i < n1 || j < n2)
  {
    if (i == n1)
      temp[k++] = sechalf[j++];
    else if (j == n2)
      temp[k++] = data[i++];
    else if (data[i] <= sechalf[j])
      temp[k++] = data[i++];
    else
      temp[k++] = sechalf[j++];
  }
  for (i=0;i<n1+n2;i++)
    data[i] = temp[i];
  delete [] temp;
}
```

Quick Sort

- **partition** array around a pivot/divider (O(N))
  - pivot should evenly divide array, but hard to find median (merge sort would be faster at this point)
    - so choose first value or a random pivot
    - recursively sort each half
  - similar relationship to merge sort: pick a pivot (constant run time), recursively sort
  - other possible *improvements* (Sedgewick):
  1. find **median** of a small sample for the pivot, eg. first 3 items
     - but this is bad if array is already sorted
     - instead, median of 3 (first, middle, last)
       - 3 is the best compared to 5, 7, etc. values
       - not necessarily first, middle, last (vulnerable to forcing an $O(N^2)$), can be random
  2. have quick sort abandon sub-arrays of size 9 or fewer
     - but the various pivots are still in their right place!
     - then call **insertion sort** / **heap sort** across the entire array!
     - items are no more than constant distance (9) out of place, so O(N)

    for insertion sort!
    - 9 is the sweet spot
  3. STL uses a *variation* on quick sort called **introsort**:
    - deep recursion happens if there are bad pivot choices
    - when depth exceeds some limit (2logN)
      * sort stops using quick sort for this *sub-region*, instead uses **heap sort**
    - completely avoid O(N^2) runtime!
    - *always* O(NlogN)
- *best case*: O(NlogN)
- *average case*: O(NlogN) - depends on the split, but *better* constant of proportionality than merge sort
- *worst case*: O(N^2) - every pivot is the worst possible pivot, least or biggest value (mostly sorted or reverse order)
  - $T(N) = O(N) + T(N-1)$ ~ O(N^2)
- good sort to use if occasional O(N^2) is acceptable
  - but requires *random access* (so only for arrays/vectors, **not** lists)!
    * picking pivot requires *random access*
- example sort:

```
[5]3  7  6  1  8  2  4 # partition array around first pivot
      4  2        6  7 # swap elements that are on wrong side from either end
1  3  4  2  [5]8  6  7 # put pivot in corresponding spot when the ends meet
[1]3  4  2             # sort left half
1  [3]2  4             # one item is a base case, sort right sub-half
1  2  [3]4
               [8]6  7 # sort right half
               7  6 [8]
               6  7    # two items is another base case
1  2  3  4  5  6  7  8
```

- Implementation:

```
void QuickSort(int Array[], int First, int Last)
{
  if (Last - First ≥ 1 )
  {
    int PivotIndex;
    PivotIndex = Partition(Array,First,Last);
    QuickSort(Array,First,PivotIndex-1); // left
    QuickSort(Array,PivotIndex+1,Last);  // right
  }
}
```

```
int Partition(int a[], int low, int high)
{
  int pi = low;
  int pivot = a[low];
  do
  {
    while ( low <= high && a[low] <= pivot )
      low++;
    while ( a[high] > pivot )
      high--;
    if ( low < high )
      swap(a[low], a[high]);
  }
  while ( low < high );
  swap(a[pi], a[high]);
  pi = high;
  return(pi);
}
```

Heap Sort

- using heaps
- approach: we can sort by inserting all the items into a **maxheap**, and then extracting them one by one
- instead, build the maxheap *in place*...

1. **interpret** unsorted array into a complete binary tree (in array notation, not neccesarily a maxheap right now)
    - starting from the bottom right item, make a heap out of the current subtree by *trickling* down
    - then work upwards through tree / *backwards* in the array
    - but bottom most leaf nodes are already heaps of their own (only one item)
        - so we can start the recursion at N/2 - 1 (in the array)
    - want to sort in ascending order, but we made a *maxheap*?
2. **remove** each item:
    - swap position 0 of array with N-1, reduce N by 1, use same trickle down approach as extracting an item
        - just like when extracting from a maxheap, but swapping an item instead of deleting
    - when down to two items, just swap in order

- O(NlogN), not as good constant of proportionality as quick sort

- use this if we only need a certain number of items in order (can stop heap sort early)

Stability of Sorts

- a **stable** sort maintains items that have equal value in the original order they were in
  - only O(NlogN) sort that guarantees this is **merge sort**

Table 2: Comparing Sorts

Sort	Stability	Big(O)	Notes
Selection Sort	Unstable	Always $O(N^2)$	Simple to implement, works with linked lists. Minimizes number of item-swaps (good if swaps are expensive)
Insertion Sort	Stable	O(N) when sorted or nearly sorted, $O(N^2$ otherwise)	Simple to implement, works with linked lists.
Bubble Sort	Stable	O(N) when sorted or nearly sorted, simple to implement	Works with linked lists. (slow, not recommended)
Shell Sort	Unstable	~$O(N^{1.25})$	OK with linked lists. Used in embedded systems due to fixed RAM usage.
Quick Sort	Unstable	O(NlogN) average, $O(N^2)$ for nearly, sorted, reverse sorted or repeated values (but O(NlogN with introsort, and best constant of proportionality)	Limited pivot choice with linked lists, can be parallelized across multiple cores, can require O(N) slots of RAM for recursion (worst) or O(logN) (average).

Sort	Stability	Big(O)	Notes
Merge Sort	Stable	O(NlogN) always	Works with linked lists, can be parallelized across multiple cores. Used for sorting data on disk (external sorting), requires O(N) slots of extra memory for merging. Requires extra storage to have a good constant of proportionality.
Heap Sort	Unstable	O(NlogN) always	Used in embedded systems due to low RAM usage / performance.

Inheritance

Drawing with Shapes

```
class Shape // Circles and Rectangles are Shapes
{
  virtual void move(double xnew, double ynew);
  virtual void draw() const; // now, program will decide at runtime which
                             // appropriate function to call
  double m_x;
  double m_y;
};

void Shape::move(double xnew, double ynew) // same move() function for every shape
{
  m_x = xnew; // compilation error if m_x and m_y aren't declared in the class Shape
```

```
  m_y = ynew;
}

class Circle : public Shape // tells compiler a Circle is a kind of Shape
{
  // void move(double xnew, double ynew); // inherits move() function from Shape
  virtual void draw() const;             // good practice to denote virtual here
  // double m_x;                         // inherits data members from Shape
  // double m_y;
  double m_r;
};

class Rectangle : public Shape
{
  // void move(double xnew, double ynew); // inherits move() function from Shape
   virtual void draw() const;            // good practice to denote virtual here
   virtual double diag() const;
  // double m_x;                         // inherits data members from Shape
  // double m_y;
  double m_dx;
  double m_dy;
};

void Shape::draw() const // how do we call the derived class's draw()?
                         // declared virtual in prototype
{                        // for now, we have to implement this generic draw() function
  ... draw a cloud ...   // generic draw action for any shape...
}
void Circle::draw() const { draw a circle }
void Rectangle::draw() const { draw a rectangle }

// etc for other shapes...
// let's draw a picture...

// ???* pic[100]; Wouldn't work, different shape classes
// Circle* ca[100];
// Rectangle* ra[100];
Shape* pic[100];     // can hold Circles AND Rectangles
pic[0] = new Circle; // how do we tell compiler Circle is a type of shape?
                     // converts Circle pointer to a Shape pointer
                     // pic really does contain pointers to shapes
pic[1] = new Rectangle;
```

```
pic[2] = new Circle;

for (int k = 0; k < ...; k++)
  pic[k]->draw();   // assumes all shapes have a draw() function
// for( int k = 0; k < ...; k++)
//   ca[k]->draw();
// for( int k = 0; k < ...; k++)
//   ra[k]->draw();

void f(Shape* x)    // for any shape now
{                   // converts Circle reference to Shape reference
  x->move(...,...); // move() works with all shapes
  x->draw();        // but draw() is different for every type of Shape!
}
// void f(Circle* x)
// {
//   x->move(...,...);
//   x->draw();
// }
// void f(Rectangle* x)
// {
//   x->move(...,...);
//   x->draw();
// }

double Rectangle::diag() const
{
  return sqrt(m_dx*m_dx + m_dy*m_dy);
}
double Square::diag() const // a reason to override a virtual function:
{                           // a more efficient implementtation
  return m_dx * sqrt(2);
}
```

- lots of **repetition**, some of these functions should work for every shape
  - our move() function will take the same parameters for *all shapes*, just absolute x and y coordinates
  - have to use multiple arrays and functions:
    - array for **each** shape type, needs a *family of functions* for different shapes
  - error prone to add a new shape!
- how to reduce repetition and make it easier to add new objects / classes?

  - find a **generalization**!
  - circles and rectangles are all shapes
  - need approach that *works for all shapes*

Inheritance

- this is **inheritance**!
  - **goals** of inheritance:
    - **reuse**:
      - every public function from base class is automatically reused in derived class
      - **not** private members, however!
    - **extension**:
      - *adding new* functions or data to a derived class
      - however, unknown to the base class!
    - **specialization**:
      - **override** existing functions from base using the virtual keyword in declaration
      - can still call base function using :: operator
  - *general steps* for inheritance:
    - figure out what to **represent** (bunch of shapes)
    - define base class with functions *common to all* derived classes (area(), draw())
    - write derived classes with **specialized** versions of each common function
    - using **polymorphism**, we can access derived variables with base ptr / reference
    - remember to define virtual destructor in base class
- generalization: Circles and Rectangles are a **subclass** of Shape
  - in c++, they are both *derived classes* of Shape
- Shape is a **superclass**
  - in c++, Shape is a *base class*
- a derived class pointer is *automatically converted* to a base class pointer (*upcast* is automatic)
  - Derived* => Base* (pointer to base within the specific derived class)
  - Derived& => Base& (same for references)
  - Derived => Base (c++ allows this, ie. *slicing*)
    - eg. Shape s(C); s = c;
    - converts to the base object **within** the derived object
    - **BUT** we lose the derived object
    - preferable to use pointers or references
  - **NO** automatic conversion from Base* => Derived*
    - *downcast*, have to use static_cast<SomeDerivedClass*>

    * must be sure our conversion is meaningful:
        · undefined behavior if base ptr doesn't actually point to a derived object of that type
- can't have **contrary** relationship where two classes can't be derived from *each other*!

Inheritance in Memory

- how can compiler access data members in a base class then?
  - every derived object has an **embedded** base object within it
  - compiler calculates the **layout** of the derived objects in memory
  - this is how the *automatic conversion works* between base and derived classses
  - **note**: *can't access* a derived class's members given a pointer to its base class

Functions within Base and Derived Classes

- how to make sure every shape can be drawn and moved?
  - add to **declaration** of Shape
  - but the **implementation** for some functions are *different for each type* of shape...
  - draw() might be different, but move() would act the same for different types
    * don't have to repeat the implementation of move() for every type of shape!
      · just implement move() **IN** shape, and remove declaration of move() from the derived classes
      · the derived classes **inherit** the move() function from Shape!
    * we can do the same for the data members that are the *same for all shapes*
  - just have to implement what makes derived classes **different** from their base classes
    * eliminates *duplication of code*!

Virtual Functions

- what about for **draw**()?
  - different implementation for every shape...
  - but *still needs to be declared within* Shape, so that f() would compile!
    * must guarantee *all shapes have* a draw()
  - for now, we need a **generic** draw() for just a Shape
  - how do we call the derived class's specific draw() functions?

    * tell compiler to make the decision at runtime **NOT** compile-time (would call generic draw())
      · *static binding*: compile-time binding, call **generic** function no matter what (cheaper!)
      · *dynamic binding*: runtime binding, call **appropriate** function depending on object
  - different binding options for different languages
  - **usually**: dynamic by default, have to indicate static
  - **c++**: static by default, have to indicate dynamic (keyword **virtual** in **declaration**)
    * optional to repeat virtual in all the implementations of derived classes' functions
- *virtual functions* allow base class functionality to be redefined in derived classes
  - can still access base class functions using base:: prefix
    * eg. Person::talks()

Overriding Functions

```
class WarningSymbol : public Shape
{
  void move(double xnew, double ynew);
  ...
}

void WarningSymbol::move(double xnew, double ynew)
{
  // move(xnew, ynew); // won't work, recursive call
  // this->Shape::move(xnew, ynew);
  Shape::move(xnew, ynew);
  ... move like a Shape
  ... flash 3 times
}

WarningSymbol ws(...);
ws.move(...);  // warning symbol DOES flash! compiler calls the immediate move()
f(ws);         // warning symbol does NOT flash! calls Shape's move() function!
Shape* sp = &ws;
sp->move(...); // warning symbol does NOT flash!

void f(Shape& x)
{
```

```
  x.move(...);
}
```

- what if we then have a new Shape that **doesn't** move like all the other shapes?
  - eg. warning symbol that flashes 3 times
  - let's write a new move() that calls Shape's move()
- if we write a new move() function:
  - works correctly when called as a WarningSymbol object
  - but when called after being **converted** to a Shape, *will not flash*!
- **redefining** base class move as virtual **would** fix the problem
  - **BUT** could be expensive to redefine a large program like this
- in our example case, let's redefine move() as virtual

Virtual Functions Demystified

```
Shape sp;
if (...)
  sp = new Rectangle(...)
else ...    // sp is some other shape

sp->draw();     // how does compiler know which draw() to call?
                // effectively: call function sp->vptr[1] points to
// sp->diag(); // remember, we can't do this, can only call Shape functions
                // with a Shape ptr...
```

- **polymorphism** refers to the same function call leading to multiple actions
  - eg. when using a base pointer or a base reference to access a derived object
- compiler makes a table holding virtual functions (ie. *virtual table*, vtbl)
  - **Shape**'s vtbl holds move() and draw()
    * the position of these functions in vtbl is the **same** for all derived classes
  - **Rectangle**'s vtbl holds move(), draw(), and diag()
    * the move() *points to Shape's* move() since Rectangle doesn't override it
- but how do we know which slot in vtbl to go to?
  - the compiler adds an *extra data member* whenever classes have a virtual function
    * *virtual pointer*, vptr, that points to vtbl!
    * all constructors also **initialize** the vptr to point to the correct vtbl
- this implementation of virtual functions *doesn't require recompilation* when more Shape types are added

Pure Virtual Functions

```
class Shape
{
  virtual void move(double xnew, double ynew);
  // virtual void draw() const;  // still necessary to declare here!
  virtual void draw() const = 0; // no longer need generic draw() implementation!
                                 // pure virtual function (Shape now an abstract class)
  double m_x;
  double m_y;
};
// void Shape::draw() const
// { // for now, we have to implement this generic function
//   ... draw a cloud ...
// }
```

- how do we *get rid* of the implementation of this generic, unecessary draw() function?
  - ie. avoid writing "dummy" logic
- use *pure virtual* function that points to null in the vtbl
  - no longer providing an implementation that is automatically inherited
- what happens if we call draw() with a Shape that **isn't** part of a derived object?
  - now, it is a *compilation error* to create a Shape!
- if a class contains *at least one* pure virtual function:
  - this is now an *abstract class* / *abstract base class* (ABC)
    - **NOT** allowed to create objects of that class type!
    - not an issue:
      - many other abstractions eg. shape, mammal, many base classes
  - however, can still have references / pointers to a Shape!
    - can also still *construct* abstract class within another!
- *derived classes* can also be abstract
  - eg. ClosedFigure inheriting from Shape with pure virtual function fillColor()
  - ClosedFigure will inherit the pure virtual function draw() from Shape!
- *failing to declare* pure virtual function in a derived class means it would also be abstract
- ABC's also **force** user to implement certain functions to prevent bugs

Construction

```
class Shape
{
 public:
```

```
  Shape(double x, double y);
 private:
  double m_x;
  double m_y;
};
Shape::Shape(double x, double y)
  : m_x(x), m_y(y)
{}

class Circle : public Shape
{
 public:
  Circle(double x, double y, double r);
 private:
  double m_r;
};
Circle::Circle(double x, double y, double r)
// : m_x(x), m_y(y), m_r(r) // wrong! trying to access private members of Shape
  : Shape(x, y), m_r(r)
{}
```

- steps of construction:
  1. construct the base object (no matter where its listed)
  2. construct each data member, consulting the *member initialization list*
  - if not listed:
    * built-in type: left uninitialized
    * class type: default-constructed (having no default constructor is an **error**)
  3. execute *body of constructor*
- steps of destruction:
  1. execute body of destructor
  2. destroy data members / objects
  3. destroy the base object

Destruction

```
class Shape
{
 public:
  ...
  virtual void draw() = 0;
  // virtual ~Shape() = 0; // should be virtual
```

```
  virtual ~Shape();
}

Shape::~Shape()
{}

class Polygon : public Shape
{
 public:
  ...
  virtual ~Polygon();
 private:
  ...
  Node* head; // collection of coordinates
}
****
Shape *sp;
if (...)
  sp = new Circle(...);
else
  sp = new Polygon(...);
delete sp; // calls Shape's destructor! leads to memory leak
```

- base class's destructor must be declared virtual to call the correct destructor!
- compiler error that there is *no implementation* for the base object destructor
  - **can't** be a pure virtual function:
    - destructor for all derived classes **call** this destructor
  - but since we declared it, destructor is *no longer* compiler generated
- *in general*: if a class is designed as a base class, declare its destructor as virtual and implement

Function and Class Templates

Function Templates

- *multiple* functions for the *same* operations with *different* types
  - we can overload these functions
  - eg. minimum(int a, int b), minimum(double a, double b)
    - same algorithm for finding the minimum…

        * but *different machine language* for comparing ints vs. doubles (different byte sizes)
        * but source code is **identical**!
- instead, can we tell compiler the **pattern** for the algorithm?
    - pattern for manufacturing a function
    - use **template** syntax in c++
- multiple templatized function calls of the same type call same function for that type
- can also have multi-type templates
- *steps for templates*:
    - match some template function (doesn't consider conversions)
        * "*template argument deduction*"
            · have to match the type *exactly*, except for matching to const in template (simple conversion)
        * at least one formal parameter must be defined by the template
    - instantiated template must compile
    - compiled code must function properly
        * eg. comparing c-strings with `<` only compares the pointers, undesired result
            · instead would need another minimum() using strcmp(a, b)
- non-template functions *takes precedence* over a template function
    - specific case over general case
- should pass by *constant reference* instead of value to deal with expensive types!
    - because it's possible to take any type in the template
- template functions **must** be declared and implemented in header files!
    - use #include to use the templated functions

```
template<typename T> // typename and class are interchangeable (besides convention)
T minimum(const T& a, const T& b) // more efficient to pass by const reference
{
  if (a < b)
    return a;
  else
    return b;
}
template<typename T1, typename T2>
?? minimum(T1 a, T2 b) // works if called with int double, double int, int int, etc.
{
  if (a < b)
    return a;
  else
    return b;
```

```
}

int k = 3;
double x = 3.0;
minimum(k, 3);    // compiler manufactures code for minimum() with ints
minimum(x, 3.14); // same for doubles
minimum(k, x);    // no matching function template! doesn't consider conversions!
                  // now would match second template function (multi-type)
                  // but could return an int instead of a double depending
                  // on order of parameters
                  // this is impractical...
Chicken c1, c2;
// minimum(c1, c2); // comparison operator not overloaded for chickens!
```

Useful Construction Syntax

```
template<typename T>
T sum(const T a[], int n) // could sum up ints, chars... strings(?)
{
  // T total = 0;         // no constructor for strings with just an int...
  // T total;             // would work for strings now, starts as empty string
                          // but for primitives, would be uninitialized
  T total = T();          // calls default constructor for strings,
                          // appropriate value for built-in types (0 false, nullptr)
  for (int k = 0; k < n; k++)
    total += a[k];
  return total;

  string x(10, '*');
  cout << x;               // if only using an object once, can use temporary object
  cout << string(10, 'x'); // string of 10 stars
  double(x);               // can also use this syntax for built-in types with
                           // appropriate value (0, false, nullptr)
}
```

Template Errors

- used to have vague compile error messages when using templates
  - hard to track down errors across multiple functions, eg. code example below
    - only pointer is being passed
  - we *never see* the function compiled from template

- now compilers provide a *traceback* of function calls

```
template<typename T>
void g(T x)
{
  T y(x);
  minimum(x, y);
}
template<typename T>
void f(T x)
{
  g(x);
}
int main()
{
  f(i); // int
  f(d); // double
  f(c); // chicken, compiler error!
}
```

Class Templates

- making a stack class:
  - could use a type alias for different types
  - but what about different stack types in the *same* program
  - **class templates** solve this
- write a **pattern** for manufacturing classes!
- similar syntax:
- special syntax for returning an internal struct!

```
template<typename T>
class Stack
{
 public:
  Stack();
  Stack(const Stack<T>& other);
  Stack<T>& operator=(const Stack<T>& rhs);
  void push(const T& x);
  T top() const;
  int size() const; // should stil return an int!
  ...
  SomeStruct returnStruct();
  SomeStruct* returnStructPtr();
  ...
```

```
 private:
  T m_data[100];
  int m_top;

  struct SomeStruct
  {
    T temp;
    int tempNum;
    ...
  }
}

template<typename T>
inline T Stack<T>::someInlineFunc(T a) {}

template<typename T>
Stack<T>::Stack() : m_top(0) {}

template<typename T>
Stack<T>::Stack(const Stack<T> other) {}

template<typename T>
Stack<T>& Stack<T>::operator=(const Stack<T> rhs) {}

template<typename T>// for every function, have to restate the template
void Stack<T>::push(const T& x) {} // works on a stack of T's

template<typename T>
T Stack<T>::top() const {}

template<typename T>
typename Stack<T>::SomeStruct Stack<T>::returnStruct() {}

template<typename T>
typename Stack<T>::SomeStruct* Stack<T>::returnStructPtr() {}

int main()
{
  Stack<int> si;       // manufactures constructor, destructor
  si.push(3);          // known to be a stack of ints,
                       // manufactures push function with ints
  Stack<Coord> sc;     // manufactures constructor, destructor
```

```
                            // error if Coord has no default constructor!
  sc.push(Coord(3,5)); // manufactures push with Coords
}
```

Template Specialization

```
template<> // can define specific behaviors for chars, such as uppercase / lowercase
class Pair<char>
{
  // must redefine entire class!
  // (also possible to only specialize certain functions)
  ...
}
```

Appendix

File Input / Output

- #include
- #include
  - defines std::ofstream (output file stream)
  - defines std::ifstream (input file stream)

Output

```
ofstream outfile("results.txt"); // outfile is a name of our choosing.
                                  // (forward slash for windows)
if ( ! outfile )          // Did the creation fail?
{
  cerr << "Error: Cannot create results.txt!" << endl;
  ... return with failure ...
}
outfile << "This will be written to the file" << endl;
outfile << "2 + 2 = " << 2+2 << endl;
```

Input

```
ifstream infile("data.txt");    // infile is a name of our choosing
if ( ! infile )              // Did opening the file fail?
{
  cerr << "Error: Cannot open data.txt!" << endl;
  ... return with failure ...
}

// read an integer:
int k;
infile >> k;
  // If you want to consume and ignore the rest of the line the
  // number is found on, follow this with
infile.ignore(10000, '\n');

// read a std::string:
std::string s;
infile >> s;         // read the next word into s
  // or
getline(infile, s);  // read a whole line into s

// read the next character from the input,
// whether it's a letter, blank, newline, or whatever:
char c;
infile.get(c);

// check for end of the input file:
  // Example 1
std::string s;
getline(infile, s);
if ( ! infile)
  cerr << "End of file when trying to read a string" << end;

  // Example 2 - read and process each line of a file until end
std::string s;
  // getline returns infile; the while tests its success/failure state
while (getline(infile, s))
{
    ... process s
}

  // Example 3 - read and process each character of a file until end
char c;
```

```
  // get returns infile; the while tests its success/failure state
while (infile.get(c))
{
  ... process c
}

  // Example 4 - read and process each integer in a file until end
int k;
  // operator>> returns infile; the while tests its success/failure state
while (infile >> k)
{
  ... process k
}

// while(!infile.eof())
// does not return true until after attempt to read past eof is made!
```

Stream Parameters

```
void greet(ostream& outf)// outf is a name of our choosing
{                         // not ofstream, pass by reference
                          // (can reference cout or a ofstream attached to file)
  outf << "Hello" << endl;
}
int main()
{
  ofstream outfile("greeting.txt");
  if ( ! outfile )
  {
    cerr << "Error: Cannot create greeting.txt!" << endl;
    return 1;
  }
  greet(outfile); // writes Hello to the file greetings.txt
  greet(cout);    // writes Hello to the screen
}

int countLines(istream& inf)   // inf is a name of our choosing
{
  int lineCount = 0;
  string line;
  while (getline(inf, line))
  lineCount++;
```

```
  return lineCount;
}

int main()
{
  ifstream infile("data.txt");
  if ( ! infile )
  {
    cerr << "Error: Cannot open data.txt!" << endl;
    return 1;
  }
  int fileLines = countLines(infile);  // reads from the file data.txt
  cout << "data.txt has " << fileLines << " lines." << endl;
  cout << "Type lines, then ctrl-Z (Windows) or ctrl-D (UNIX):" << endl;
  int keyboardLines = countLines(cin); // reads from keyboard
  cout << "You typed " << keyboardLines << " lines." << endl;
}
```

Object Oriented Design Steps

- generally, two phases:
  - determine overall class design
  - determine class data structures / algorithms
- *class design*:
  1. classes and objects necessary
  2. outward-facing functionality
  3. data each class holds
  4. how they interact
- *class design steps*:
  1. identify potential **classes**
  - *nouns* in the spec.!
  - eg. calendar, appointments, time-slot, password, start-time, end-time, participants
    * narrow down into classes (calendar, appointment)
  2. identify **operations**
  - what actions need to be performed in spec (*verbs*!)
  - eg. add new appnt., remove existing, check other users' calendars, supply password
    * associate actions with classes (functions)
  3. determine *relationships & data*
  - *general relationships*:
    * Class A **uses** objects of class B, not necessarily contains

        * Class A **has-a** (contains) objects of class B (**composition**)
        * Class A **is-a** specialized version of class B
        * this determines private data & inheritance
        * eg. calendar contains appointments, has a password, uses other calendars
    4. determine **interactions**
    - determine how each class interacts with the others
    - come up with *use-cases*:
        * eg. user wants to add (locate, update) an appointment, determine if they have an appointment
    - class design is an **iterative** process!
- *tips*:
    - **avoid** using dynamic cast to identify common types of objects
        * instead add functions to check for various *classes of behaviors*
        * eg. if (p->requiresOilToOperate()) ...
    - **avoid** defining specific isClass() functions for every object
        * instead add functions to check for various *common behaviors*
    - **avoid** duplication of member variables in related subclasses
        * instead move to base class with accessor / mutator methods
    - **never** make data members public or protected
        * class constants may be, however
    - **never** make a function public if only used in class that holds it
        * instead private or protected
    - **never** return list / vector / iterator / pointers to private objects
        * instead do all the processing within the class if action needs to be taken

Discussion

Administrative

- Midterms 1/30, 2/26
- Final 3/16
- Start early, develop incrementally, read what you write !!!

1.18.19

- Can use Valgrind tool to detect memory leaks
- Class Composition
    - when class contains member variables that are objects

  - Order of Construction (*inside to outside*)
    * member variables constructed in order (in order of the *member variable listing*, member initialization list doesn't matter)
    * then current class constructor
  - Order of Destruction (*outside to inside*)
    * current class destructor called first
    * member variables destructed in **reverse** order
  - class composition != class **inheritance**
- Copy Constructors
  - *shallow copy* just copies all data members
  - *deep copy* (redefine default copy constructor) is needed in the cases of pointers / when not **SHARING** the same adresses / using dynamic memory
  - similar to writing assignment operator (make sure to delete current dynamic memory and return *this at the end)
- Data Structures
  - 3 functions
    * how to store and organize
    * how to add and remove data
    * how to apply functions (eg. search data)
  - add, contain, remove...
  - Pros / cons for every structure and differing efficency / complexity

2.1.19

- **Linked Lists**:
  - made up of Nodes with **value** and **pointer**
  - *head pointer* points to first term
  - loop-free
  - **variations**:
    * doubly linked, sorted, circular
  - **operations**:
    * insertion, search, removal
  - **pros**: efficient insertion, flexible memory, simple implementation
  - **cons**: complex searching and delete