User-Defined Simple Data Types

- data type
  - a set of values and a set of operations on those values
- for example, integer
  - set of values: approx. -2GB to approx. +2GB
  - set of operations: =, +, -, *, /, %, etc.
- compiler enforced restruictions on usage values/operations
- enumeration type
  - user define custom data type
  - user defined set of ordered values
  - limited set of operations: +, <, <=, >, >=, ==
- enumeration syntax
  - declation: enum typeName { value1, value2, ... };
  - example: enum allowedSelections { ROCK, PAPER, SCISSORS } ;
  - usage: answer = ROCK;
  - usage: if (answer == PAPER) { }
  - example 2: enum allowedSelections { 1ST, 2ND, 3RD } ;
  - *note*, 1ST is < 2ND since the enum values are ordered
- Functions
  - intrinsic (pre-defined)
  - user defined
  - prototype
  - return
  - actual arguments vs formal arguments
  - scope
  - call by value
  - call by reference
- C++ Random Numbers
- the **rand()** function will return an integer random number
  - between 0 and RAND_MAX (a system defined maximum)
- the **drand48()** function will return a double random number
  - between 0.0 and 1.0 such that 0.0 <= drand48() < 1.0
- for integers (typical use), the **rand()** is easier to use
  - may need to be scaled for certain applications
- to scale a random number to a range from 1-100, the modulo (remainder after division) can be used
  - for example
  - int prob = rand() % 100 + 1;
- C++ Switch Statement

    - the switch statement chooses statements to execute depending on an integer value
    - the same effect can be achieved with a series of cascading if statements
        - but in some cases the switch statement is easier to read
        - some compilers will produce more efficient code
    - the idea is that this is easier to understand and debug than a long series of nested if/elseif statements.
    - the **break** statement exits from the switch statement.
        - if there is no break at the end of a case
            - execution continues in the next case
            - which is typically an error
    - an example is as follows:
        - *expr* below evaluates to an integer
        - c1, c2, c3, and c4 are constant values
- Function Overloading
    - Creating multiple functions with the same name
    - Two functions can have a different formal parameter list
        - different number of formal parameters
        - if number of formal parameters is same
            - different data types
    - function signature
- Default Parameters
    - overview
    - must specify the value of a default parameter in the prototype
    - if no value is provided, default value is used
    - any default parameters must be the rightmost parameters
    - for > 1 default parameter and a value is not specified -> must omit all of the arguments to right
    - default values can be constants
    - example
        - assume: a, b are int, ch is char, d is double
- Object Oriented Approach
    - OO defines the data structure and the operations (functions) that can be applied to the data structure
    - object includes both data and functions
    - can create relationships between one object and another
    - objects can inherit characteristics from other objects
    - modules do not need to be changed when a new type of object is added
        - thus, object-oriented programs easier to modify
    - objects are treated as "black boxes" which send and receive messages

- Classes and Data Abstraction
    - classes

        - a class is an expanded concept of a data structure
        - instead of holding only data, a class can hold both data and functions
        - an object is an instantiation of a class.
        - in terms of variables, a class would be the type, and an object would be the variable
    - class definitions
        - classes are declared using the keyword class
- Classes
    - if class member is variable -> declare as usual
        - can not initialize
    - if class member is function -> use prototype to declare the function
        - function can directly access other members (variables/functions)

- Class Format
    - basic format
    - the `className` is the identifier for the class
    - the body of the declaration contains class members
        - can be either data or function declarations
    - similar to the declaration of data structures
        - except now expected to include also functions and data
        - specifier -> an access specifier is **private**, **public** or **protected**
            - private members of a class are accessible only from within other members of the same class or from their friends
                - *note*, friend functions addressed later
            - protected members are accessible from members of the same class and from their friends, but also from members of their derived classes
            - public members are accessible from anywhere where the object is visible
        - no required order for specifiers

- Declarations
    - class definition
    - object instantiation (of class)
        - functions -> memory use (one copy)
        - variables -> memory use (copy per object)
    - scope
        - scope rules for objects same as other variables
        - private, protectedm and public control scope of memebers.

- Class Operations (built-in)
  - assignment (=)
  - member access (.)

- Example
  - clockType example (from text)

- Constructors
  - constructor with parameters
  - constructor without parameters
    - default constructor
  - constructors have no type
    - not void, just no type
  - multiple constructors allowed
    - function overloading
    - must have different signatures
  - constructor execute automatically

- Building executable
  - compilation to object file
    - g++ -c srcFile1.cpp
    - g++ -c srcFile2.cpp
  - link object files, create executable file
    - g++ -o projName srcFile1.o srcFile2.o
  - for example (on asst #4)
- Unified Modeling Language (UML)
  - class name
  - member variables
    - variable name, followed by type
    - a **+** indicates public variable
    - a **-** indicates private variable
    - a **+** indicates protected variable
  - member functions
    - function name, argument list, followed by type
    - a **+** indicates public function
    - a **-** indicates private function
    - a **+** indicates protected function

- Review

- Object Oriented Approach
  - OO defines the data structure and the operations (functions) that can be applied to the data structure
  - OO uses "block box" approach
- Declaration
  - class definition
  - object instantiation (of class)
  - object memory allocation
    - function memory use (one copy)
    - variable memory use (copy per object)
  - scope
    - internal to class
    - external to class
- Class Operations (built-in)
  - assignment (=)
  - member access (.)

Scope Resolution Operator

- The scope resolution operator helps to identify and specify the context to which an identifier refers, particularly by specifying a namespace
- Classes and Data Abstraction
  - Classes
    - a class is an expanded concept of a data structure
    - instead of holding only data, a class can hold both data and functions
    - an object is an instantiation of a class.
    - in terms of variables, a class would be the type, and an object would be the variable
  - Abstraction
    - separating the design/implementation details from it usage
  - Abstract Data Type (ADT)
    - A data type that separates the logical parameters from the implementation details.
    - ADTs have three key associations
      - ADT name, called type name
      - set of variables in the ADT, called the domain
      - set of operations on the data
  - Classes are how C++ implemented ADTs
  - Classes / ADTs support "information hiding"
    - allow access to some information

        - hide or disallow access to other information

- Classes
    - if class member is variable -> declare as usual
        - can not initialize
    - if class member is function -> use prototype to declare the function
        - function can directly access other members (variables/functions)
    - Accessor and Mutator Functions
        - accessor functions -> only access member variables (no modification)
            - accessor functions typically declared 'const'
            - function declared 'const' can only call other 'const' functions
        - mutator functions -> modifies member variables

- Constructors
    - constructor with parameters
    - constructor without parameters
        - default constructor
    - constructors have no type
        - not void, just no type
    - multiple constructors allowed
        - function overloading
        - must have different signatures
    - constructor executes automatically
        - default parameters
        - provided parameters
        - default parameters
    - constructor does **not** automatically initialize variables
        - you add code to initialize the variables (as required based on specific parameters of the program)

- Example
    - clockType example (from text)
    - accessor functions
    - mutator functions
    - pass-by-reference
    - pass-by-value

- Building executable
    - compilation to object file

        - g++ -c srcFile1.cpp
        - g++ -c srcFile2.cpp
    - link object files, create executable file
        - g++ -o projName srcFile1.o srcFile2.o
    - for example (on asst #4)
- Review -> Classes and Data Abstraction
    - Classes
        - a class is an expanded concept of a data structure
        - instead of holding only data, a class can hold both data and functions
        - an object is an instantiation of a class.
        - in terms of variables, a class would be the type, and an object would be the variable
    - Abstraction
        - separating the design/implementation details from it usage
    - Abstract Data Type (ADT)
        - A data type that separates the logical parameters from the implementation details.
        - ADTs have three key associations
            - ADT name, called type name
            - set of variables in the ADT, called the domain
            - set of operations on the data
    - Classes are how C++ implemented ADTs
    - Classes / ADTs support "information hiding"
        - allow access to some information
        - hide or disallow access to other information

- UML Documentation
    - UML -> Unified Modeling Language
    - Key fields
        - class name
        - variables
        - functions
    - field modifiers
        - private variables/functions indicated with '-' sign
        - public variables/functions indicated with '+' sign
    - industry/academic standard
    - supported by software engineering tools
    - supports graphical representation for classes

- Class Variables - Arrays
    - generally uses default constructor
    - can set individually during array declaration

        - className varName[arrSize] = { varName(), varName(), ... }
        - index **not** specified
        - index is by position
        - while possible, generally not done

- Static vs Automatic Variables
    - static vs automatic discussed in text (chapter 7)
    - static, exists between function calls
    - automatic (non-static) created as part of function (does not exist before or after function call)
        - automatic also referred to as *stack dynamic locals*
    - static vs automatic -> trade-offs
        - memory usage
            - static variables always take up memory (even if never used)
            - automatic variables only use memory when being used
        - execution time
            - static variables do not require run-time overhead (thus faster)
            - automatic variables only use memory when being used, but require run-time overhead to create the variables dynamically when needed (and destroy when done)

- Static vs Automatic Class Variables
    - variables are automatic (non-static) by default
    - variables within a class can be declared static
    - **not** typically used much
    - when declared static, they are kept in memory
    - can mark static variables as public or private
    - for each static class variables, the compiler allocated only **one** memory location
        - regardless of how many objects are declared
        - thus each object refers to the **same** static variable(s)

- Make Utility
    - utility to perform compilations
    - handles compilation dependencies
        - automatically compiles only necessary files
            - if a source has not changed and object file is still available, no compilation needed
        - very very useful for large projects
    - makefile
        - special format

        - see provided example makefiles
    - usage
        - typical usage, type; **make**
            - review last modified dates, determines which objects need to be re-created and compiles appropriate source files, and builds executable
        - other usage, type: **make <optionalArgument>**
        - the optional argument can be used to
            - make a subset of the projected (for testing)
            - clean up some of the files (i.e., remove the object files when done)

- Classes
    - if class member is variable -> declare as usual
        - can not initialize
    - if class member is function -> use prototype to declare the function
        - function can directly access other members (variables/functions)
    - Accessor and Mutator Functions
        - accessor functions -> only access member variables (no modification)
            - accessor functions typically declared 'const'
            - function declared 'const' can only call other 'const' functions
        - mutator functions -> modifies member variables

- Constructors
    - constructor with parameters
    - constructor without parameters
        - default constructor
    - constructors have no type
        - not void, just no type
    - multiple constructors allowed
        - function overloading
        - must have different signatures
    - constructor executes automatically
        - default parameters
        - provided parameters
        - default parameters
    - constructor does **not** automatically initialize variables
        - you add code to initialize the variables (as required based on specific parameters of the program)

- Destructor
    - automatically executed when variable goes out of scope

    - only one destructor allowed
    - no type (not void, but no type)
    - name is preceded with ~ (tilda)

- Inheritance Types
    - creating new classes from existing classes
        - allows re-using existing code from base class (where appropriate)
        - allows adding new code (as needed)
        - allows re-defining functions (if needed)
        - also called "deriving" or "derivation"
    - inheritance
        - **is-a** relationship
            - hierarchical structure
            - base class
            - inherited or derived class created from "base" class
    - composition
        - also called aggregation or multiple inheritance
        - **has-a** relationship
        - aggregate classes
        - more fully addressed later
    - inheritance hierarchical example
- Base Class Member Access Specifiers (review)
    - public
        - available to inherited class and externally
    - private
        - **not** available to inherited class or externally
    - protected
        - **not** available externally
        - **is** available to inherited class

- Inherited Class Syntax
    - general syntax includes specifying the base class
    - members are defined as usual
        - example

Inherited Class Access Specifiers

- private members of base class -> still private
    - inherited class can **not** access the private members
- protected members of base class can be accessed by inherited class
    - still not accessible externally
- public members of the base class may be public (but can be changed)
    - accessible externally

- the **memberAccessSpecifier** allows over-riding of the base class specifiers
    - the *memberAccessSpecifier* apllies to the entire base class
    - thus, public members can be redefined to be private
    - allows additional restricting for otherwise accessible members in the base class
    - if the memberAccessSpecifier is marked as private
        - public members of the base class **not** be accessible externally via inherited class
        - can be access for an object of the base class (which is independent of the inherited class)
    - *note* if no member access specifier, private is the default (poor practice)
- Review -> Creating New Classes from Existing Classes
    - aka "derivation"
    - inheritance
        - 'is-a' relationship
        - hierarchical structure
        - base class
        - inherited or derived class created from "base" class

- Composition
    - **has-a** relationship
    - aggregate classes
    - can be confusion/dangerous
    - **not** allowed in Java
    - used in assignment #3
    - potential problem

- Review -> Inherited Class Access Specifiers
    - private members of base class -> still private
        - inherited class can **not** access the private members
    - protected members of base class can be accessed by inherited class
        - still not accessable externally
    - public members of the base class may be public (but can be changed)
        - accessable externally
    - the **memberAccessSpecifier** allows over-riding of the base class specifiers
        - the memberAccessSpecifier allpies to the entire base class
        - thus, public members can be redefined to be private
        - allows additional restricting for otherwise accessible members in the base class
        - if the memberAccessSpecifier is marked as private
            - public members of the base class **not** be accessable externally via inherited class
            - can be access for an object of the base class (which is independent of the inherited class)

        - *note* if no member access specifier, private is the default (poor practice)
    - example
        - the below code fragment example show the impact of the member access specifier for inheritance
- Constructors - Composition
    - constructor -> perform initializations
    - derived class have own constructors
        - derived class can not access private variables of base class
    - derived class must trigger multiple class constructors
    - the base class is referenced on the inherited class constructor definition
    - synatx for inherited class constructor
        - the **`: <baseClassName> (<formalArgumentsList>)`** is used
    - example, in the inherited class, the constructor would be

Over-Riding Base Class Functions

- a function with the same name in the inherited class will over-ride the function in the base class
- thus, a call to a function via the inherited class will automtically call the new inherited class function
- example
    - assuming baseClass has functions:
        - print(), calc(), read()
    - assuming inheritedClass has functions:
        - print(), calc()
    - if an object, myObj, is declared of type inheritedClass
        - myObj.print() will access the inherited class function print()
        - myObj.read() will access the base class function read()
- accessing Base Class Functions
    - if needed, the inherited class can access the base calss function of the same name
    - the scope resolution operated, ::, is used
    - example, assuming the baseClass and inheritedClass functions (from above)
        - myObj.calc() will access the inherited class function calc()
        - if the inherited class function calc() needs to access the base class function calc()
            - in the inherited class function calc(), accessed via **`baseClass::calc()`**
- Class Header File Conflicts
    - possible header file conflicts resulting from including the header file multiple times
        - once in base class, once in inherited class
            - thus, possibly multiple times in client code
        - multiply defined members would cause errors

            - use of ifndef
        - ifndef if not defined
    - allows something to be defined, if not already defined
    - used to ensure that if something is already defined, it is not defined again
    - example
        - if the SOME_NAME is not defined, it is defined and the entire class is also defined
        - if the SOME_NAME is already defined, nothing is done (since the name and the class are already defined)
- Review -> Inherited Class Access Specifiers
    - public, private, protected

- Review -> Over-Riding Base Class Functions
    - a function with the same name in the inherited class will over-ride the function in the base class

- Review -> Constructors - Composition
    - constructor -> perform initializations
    - derived class can have own constructors
        - derived class can not access private variables of base class
    - derived class might need to trigger multiple class constructors
    - the base class is referenced on the inherited class constructor definition
    - the `: <baseClassName> (<formalArgumentsList>)` is used
    - syntax for inherited class constructor

- Class Header File Conflicts
    - multiple inheritance -> header files conflicts
    - multiply defined members
    - use of **ifndef**
    - **ifndef** -> if not defined
    - allows something to be defined, if not defined
    - ensures if defined, not defined again

- Pointers (Ch 13)
    - memory addresses
    - declaration
        - <type> * <variable>;
    - examples
        - int * var;
        - char * var;
    - operators
        - address of -> **&**

        - indirection / dereferencing -> *****
    - example
        - int x = 42;
        - int *p;
        - p = &x;
        - cout << p << endl; -> 0x0804451a
        - cout << *p << endl; -> 42
        - *p = 4;
        - *p *= 2;
        - cout << *p << endl; -> 8
    - operations
        - given: int *p, *q;
        - equality / relational
            - p == q -> true/false
            - p != q -> true/false
        - addition / subtraction
            - p++; (or p = p+1;)
            - p--; (or p = p-1;)
            - +1 for an int is actually 4 bytes
            - +1 for a char is 1 byte
            - +1 for a double is actually 8 bytes
        - assignment
            - p = q;
        - multiplication / division
            - not allowed
            - makes no sense

- Array Names -> Pointers
    - an array names is a <b>constant pointer
    - given: int lst[10];
    - lst is a constant pointer and can not be changed
    - however, also given: int *ptr;
    - can set:
        - ptr = lst;
        - can change ptr -> ptr++;
        - can not change lst -> lst++ is illegal

- Dynamic Allocation
    - arrays can be dynamically allocated
    - uses the **new** operator
    - first, declare array pointer
        - int *arr;

    - use **new** to allocate array
        - arr = new int[100];
    - must deallocate the array when done
        - use the **delete** operation
        - delete [] arr;
- Two dimensional arrays
    - for two dimensional arrays
        - allocate array of pointers for the rows
        - allocate each row
    - must deallocate the array when done
        - use the **delete** operation for each row
        - then use delete operation for the rows array
    - example
        - allocation two dimensional array with 20 rows and 5 columns
- Parameter Passing - Objects
    - call by reference
        - void callPrint(petType& p) { p.print(); }
        - additional dogType members are ignored
    - call by value
        - void callPrint(petType p) { p.print(); }
        - copy made (since value)
        - copy is of petType
        - additional dogType members are lost (not copied)
    - using a virtual function will not fix this

- Copy Constructor
    - when declaring an object, it can be initialized by the values in another object (of same type)
    - example
        - rectangleType newRect(myRect);
        - creates new object newRect which is a copy of object myRect
        - default copy constructor provided by compiler
        - if objects contain pointer values, it does **not** allocate additional memory, just copies pointer values
        - thus, implements shallow copy
    - to provide deep copy, a copy constructor function can be written and included into the class.
    - example (in .h file)
        - className(const className& passedObject);
        - over-rides use of default copy constructor
        - can allocate additional memory and implement deep copy

- Virtual Functions

    - C++ allows passing an object of a derived class to formal parameter of the base class
    - virtual keyword
    - example
        - virtual void print();
    - binding
        - compile-time or static
            - performed when compiler
                - can not be changed
            - run-time or dynamic
                - performed during execution
                - can be changed (as needed, within limits)

- Object Pointers
    - given declarations
        - petType pet("Perry");
        - dogType dog("Rover", "German Shepard");
    - legal -> pet = dog;
        - information lost, but allowed
        - known as 'slicing problem'
    - illegal -> dog = pet;
        - information needed, not allowed
    - given declarations/statements
        - petType *pet;
        - dogType *dog;
        - dog = new dogType("Rover", "German Shepard");
    - must use -> to access functions from pointer type
        - dog->setBreed("Husky");
        - pet = dog;   // is allowed
        - pet->print();   // Name: Rover, Breed: Husky

- Virtual Destructors
    - classes with pointer variables should include destructors
        - ensures dynamically created items are deallocated
        - gameBoard used a destructor to delete the dynamically created array
    - problem -> if a derived class is passed to a formal parameter to a base class, the destructor of the base class will execute when it goes out of scope
        - this could delete things in derived class (that should not be deleted yet)
    - to address this, if the base class contains pure virtual functions, the base class destructor should also be virtual

- Pure Virtual Functions

    - no implementation in base
    - forces inherited class to have a implementation for the pure virtual function

- Abstract Classes
    - class containing one or more pure virtual functions
    - now, class is not complete
    - can not instantiate objects of the abstract class
    - example -> shape object
        - a shape is very general (can be many things)
            - the abstract object shape might include ***pure virtual*** functions to draw and move
            - can not instantiate a ***shape*** since it is no general
        - a rectangle is a specific shape
            - rectangle is derived from shape
            - due to the pure virtual functions in shape, the derived class for rectangle must include functions for draw and move
            - those functions will be specific for that rectangle object
            - it may include additional ones
    - very useful for larger projects
-
    -
      example
        - virtual void print();
    - binding
        - compile-time or static
            - performed when compiler
            - can not be changed
        - run-time or dynamic
            - performed during execution
            - can be changed (as needed, within limits)

- Parameter Passing - Objects
    - call by reference
        - void callPrint(petType& p) { p.print(); }
        - additional dogType members are ignored
    - call by value
        - void callPrint(petType p) { p.print(); }
        - copy made (since value)
        - copy is of petType
        - additional dogType members are lost (not copied)

    - using a virtual function will not fix this

- Object Pointers
    - given declarations
        - petType pet("Perry");
        - dogType dog("Rover", "German Shepard");
        - legal -> pet = dog;
            - information lost, but allowed
            - known as 'slicing problem'
        - illegal -> dog = pet;
            - information needed, not allowed
    - given declarations/statements
        - petType *pet;
        - dogType *dog;
        - dog = new dogType("Rover", "German Shepard");
    - must use -> to access functions from pointer type
        - dog->setBreed("Husky");
        - pet = dog; // is allowed
        - pet->print(); // Name: Rover, Breed: Husky
- Pointer **this**
    - a member function can directly access class variables
    - sometimes a member function can refers to the object as a whole
    - every object contains a hidden pointer to itself (this)

- Virtual Functions
    - C++ allows passing an object of a derived class to formal parameter of the base class
    - virtual keyword
    - example
        - virtual void print();
    - binding
        - compile-time or static
            - performed when compiler
            - can not be changed
        - run-time or dynamic
            - performed during execution
            - can be changed (as needed, within limits)

- Review -> Parameter Passing - Objects
    - call by reference

        - void callPrint(petType& p) { p.print(); }
        - additional dogType members are ignored
    - call by value
        - void callPrint(petType p) { p.print(); }
        - copy made (since value)
        - copy is of petType
        - additional dogType members are lost (not copied)
    - using a virtual function will not fix this

- Virtual Destructors
    - classes with pointer variables should include destructors
        - ensures dynamically created items are deallocated
        - gameBoard used a destructor to delete the dynamically created array
    - problem -> if a derived class is passed to a formal parameter to a base class, the destructor of the base class will execute when it goes out of scope
        - this could delete things in derived class (that should not be deleted yet)
    - to address this, if the base class contains pure virtual functions, the base class destructor should also be virtual

- Pure Virtual Functions
    - no implementation in base
    - forces inherited class to have a implementation for the pure virtual function

- Abstract Classes
    - class containing one or more pure virtual functions
    - now, class is not complete
    - can not instantiate objects of the abstract class
- Functions
    - function overloading
    - function default parameters

- Object Oriented Approach
    - OO defines the data structure and the operations (functions) that can be applied to the data structure
    - OO uses "block box" approach
    - Declaration
        - class definition
        - object instantiation (of class)
        - object memory allocation
            - function memory use (one copy)

            - variable memory use (copy per object)
        - scope
            - internal to class
            - external to class
    - Class Operations (built-in)
        - assignment (=)
        - member access (.)

- Classes and Data Abstraction
    - Classes
        - a class is an expanded concept of a data structure
        - instead of holding only data, a class can hold both data and functions
        - an object is an instantiation of a class.
        - in terms of variables, a class would be the type, and an object would be the variable
    - Abstraction
        - separating the design/implementation details from it usage
    - Abstract Data Type (ADT)
        - a data type that separates the logical parameters from the implementation details.
        - ADTs have three key associations
            - ADT name, called type name
            - set of variables in the ADT, called the domain
            - set of operations on the data
    - Classes are how C++ implemented ADTs
    - Classes / ADTs support "information hiding"
        - allow access to some information
        - hide or disallow access to other information

- Class Operations (built-in)
    - assignment (=)
    - member access (.)

- Constructors
    - constructor with parameters
    - constructor without parameters
        - default constructor
    - constructors have no type
        - not void, just no type
    - multiple constructors allowed
        - function overloading
        - must have different signatures

    - constructor execute automatically
    - Destructors
- automatically executed when variable goes out of scope
- only one destructor allowed
- no type (not void, but no type)
- name is preceded with ~ (tilda)
- Class Variables - Arrays
    - generally uses default constructor
    - can set individually during array declaration
        - className varName[arrSize] = { varName(), varName(), ... }
        - index not specified
        - index is by position
        - while possible, generally not done

- Static vs Automatic
    - static, exists between function calls
    - automatic (non-static) created as part of function (does not exist before or after function call)
    - trade-offs
        - memory usage
        - execution time

- Static vs Automatic Class Variables
    - variables are automatic (non-static) by default
    - variables within a class can be declared static
    - **not** typically used much
    - when declared static, they are kept in memory
    - can mark static variables as public or private
    - for each static class variables, the compiler allocated only one memory location
        - regardless of how many objects are declared
        - thus each object refers to the same static variable(s)

- UML Documentation
    - UML -> Unified Modeling Language
    - Key fields
        - class name
        - variables
        - functions
    - field modifiers
        - private variables/functions indicated with '-' sign
        - public variables/functions indicated with '+' sign

    - industry/academic standard
    - supported by software engineering tools
    - supports graphical representation for classes

- Review -> Creating New Classes from Existing Classes
    - aka "derivation"
    - Inheritance
        - 'is-a' relationship
        - hierarchical structure
        - base class
        - inherited or derived class created from "base" class

- Base Class Member Access Specifiers
    - public
        - available to inherited class and externally
    - private
        - **not** available to inherited class or externally
    - protected
        - **not** available externally
        - **is** available to inherited class

- Inherited Class Access Specifiers
    - private members of base class -> still private
    - addresses public members of base class
    - allows inherited class to change/over-ride base class specifiers
    - public
        - public members of the base class are public
    - private
        - private is default, if unspecified
        - public members of the base class are private
        - public members of the base class not accessible externally via inherited class types

- Over-Riding Base Class Functions
    - a function with the same name in the inherited class will over-ride the function in the base class

- Inherited Class Constructor Function Definitions
    - as before, functions are declared using the className, the "::", and the functionName (followed by the formal arguments)
    - the base class constructor is called on the function delcaration line
    - syntax is **:** followed by base class constructor call

Class Header File Conflicts

- multiple inheritance -> header files conflicts
    - multiply defined members
- use of "ifndef"
    - ifndef -> if not defined
- allows something to be defined, if not defined
- ensures if defined, not defined again
- Pointers
    - memory addresses
    - declaration
    - operators
    - address of -> &
        - indirection / dereferencing -> *
    - operations
        - addition / subtraction
        - assignment
        - multiplication / division
            - not allowed
    - array Names -> Pointers
        - array names are a constant pointer

- Virtual Functions
    - C++ allows passing an object of a derived class to formal parameter of the base class
    - virtual keyword
    - example
        - virtual void print();
    - binding
        - compile-time or static
            - performed when compiler
            - can not be changed
        - run-time or dynamic
            - performed during execution
            - can be changed (as needed, within limits)

- Parameter Passing - Objects
    - call by reference
        - void callPrint(petType& p) { p.print(); }
        - additional dogType members are ignored
    - call by value

        - void callPrint(petType p) { p.print(); }
        - copy made (since value)
        - copy is of petType
        - additional dogType members are lost (not copied)
    - using a virtual function will not fix this

- Virtual Destructors
    - classes with pointer variables should include destructors
        - ensures dynamically created items are deallocated
        - gameBoard used a destructor to delete the dynamically created array
    - problem -> if a derived class is passed to a formal parameter to a base class, the destructor of the base class will execute when it goes out of scope
        - this could delete things in derived class (that should not be deleted yet)
    - to address this, if the base class contains pure virtual functions, the base class destructor should also be virtual

- Pure Virtual Functions
    - no implementation in base
    - forces inherited class to have a implementation for the pure virtual function

- Abstract Classes
    - class containing one or more pure virtual functions
    - now, class is not complete
    - can not instantiate objects of the abstract class

- Operator Overloading
    - example, operator overloading
        - '+ (plus) used on integers, returns integers
        - '+ (plus) used on floats, returns floats
        - '+ (plus) used on strings, concatenates
    - most existing operators can be overloaded
    - must write header and body
    - use **operator** keyword
    - syntax for used-defined operator overloading
        - <returnType> **operator** operatorSymbol(form parameter list)
    - overloading limitations
        - can not change precedence
        - can not change associativity
        - default parameters can not be used

        - can not change the number of parameters an operator takes
        - can not create new operators
        - the meaning of how an operator works with default parameters can not be changed
        - operators that can not be overloaded
            - `.`
            - `.*`
            - `::`
            - `?:`
            - `sizeof`

- Operator Overloading -> Member Functions
    - overloading operators as member functions same as other functions
    - requires scope resolution operator, ::, and keyword *operator*
    - example, - (binary negation) operator
        - `double x=3.14;`
        - `cout << -x << endl;`
        - produces -3.14
    - example:
        - overload != (not equal) operator for complex numbers as friend function
        - overload == (not equal) operator for complex numbers as friend function

```
HEADER FILE:        friend bool complexType::operator !=
(const complexType& one, const complexType& two);
friend bool complexType::operator == (const complexType&
one, const complexType& two);    IMPLEMENTATION FILE:
bool complexType::operator != (const complexType& one, const
complexType& two)        {            return(one.realPart !=
two.realPart &&                        one.imaginaryPart !=
two.imaginaryPart);        }        bool
complexType::operator == (const complexType& one, const
complexType& two)        {            return(one.realPart ==
two.realPart &&                        one.imaginaryPart ==
two.imaginaryPart);        }
```

- Friend functions
    - a friend function is a non-member function of the class, but has full access to all class members (public, protected, and private)
    - uses **friend** keyword in header file

- Overloading << and >> operators

    - must be overloaded as friend functions
    - function prototype
        - `ostream operator << (ostream&, const className&);`
    - both parameters must be reference parameters
    - for <<
        - the first parameter must reference ostream (output stream)
        - the second parameter, usually a const reference to the class
        - the function return type is a reference to an ostream object
    - for >>
        - the first parameter must reference istream (input stream)
        - the second parameter, is a reference to the class
        - the function return type is a reference to an istream object
- Operator Overloading
    - example, operator overloading
        - '+ (plus) used on integers, returns integers, used on floats, returns floats, '+ (plus) used on strings, concatenates
    - most existing operators can be overloaded
    - must write header and body
    - use **operator** keyword
    - syntax for used-defined operator overloading
        - <returnType> **operator** operatorSymbol(form parameter list)
    - overloading limitations
        - can not change precedence
        - can not change associativity
        - default parameters can not be used
        - can not change the number of parameters an operator takes
        - can not create new operators
        - the meaning of how an operator works with default parameters can not be changed
        - operators that can not be overloaded
            - `.  .*  ::  ?:  sizeof`

- Operator Overloading -> Member Functions
    - overloading operators as member functions same as other functions
    - requires scope resolution operator, ::, and keyword *operator*
    - example:
        - overload **!=** (not equal) operator for complex numbers as friend function
        - overload == (not equal) operator for complex numbers as friend function

- Friend functions
    - a friend function is a non-member function of the class, but has full access to all class members (public, protected, and private)

- uses **friend** keyword in header file

- Overloading << and >> operators
  - must be overloaded as friend functions
  - function prototype
    - `ostream operator << (ostream&, const className&);`
  - both parameters must be reference parameters
  - for <<
    - the first parameter must reference ostream (output stream)
    - the second parameter, usually a const reference to the class
    - the function return type is a reference to an ostream object
  - for >>
    - the first parameter must reference istream (input stream)
    - the second parameter, is a reference to the class
    - the function return type is a reference to an istream object

- Exceptions
  - undesired situation during program execution
  - Some problems that arises during the execution of a C++ program
    - arithmetic
    - library functions
  - need a clean method to handle exceptions
  - some exceptions can be addressed via **if** statements
    - division by 0
  - other exceptions, not so easy
    - `cin`

- Exception Handling
  - exceptions provide a way to transfer control from one part of a program to another
  - the goal is to handle possible run-time crashes
  - allow programmer to deal with the problem
  - C++ exception handling is built upon three keywords

    - **try**
      - a try block identifies a block of code for which particular exceptions will be activated
      - followed by one or more catch blocks
    - **throw**
      - a program throws an exception when a problem shows up
      - done using a throw keyword

        - **catch**
            - a program catches an exception with an exception handler at the place in a program where you want to handle the problem
            - indicates the catching of an exception

- Exception Handling - Usage
    - execute statements within a try **{ }** block
        - no exception, no problem
        - no catch blocks are executed
    - if an exception is generated
        - program does not crash (yeah)
        - code in the applicable catch **{ }** block is executed
        - syntax: **`catch (<type> <variable>)`**
        - order of catch blocks can matter
    - exceptions can be "thrown" to a catch block
    - ensures consistent handling of errors for anomalous conditions (possibly algorithm specific) that may not generate a run-time exception
    - separates error handling code from normal code
        - better flow, improved logic, etc.
    - syntax: **`throw <expression>`**

```
try
{
     // statements that might cause exception
}

catch (<type1> <var>)
{
     // statements
}

catch (<typeN> <var>)
{
     // statements
}
```

- Example
    - example for division by zero
    - could address via if statement
        - if (divisor ==)
            - cout << "Error..." << endl;
    - could address via assert

        - assert(divisor != 0);
        - *note*, a failed assert terminates program
    - choose to use try/catch block
        - useful, simple example
        - additional want to handling invalid input
    - try/catch example
        - catches division by 0 error
        - catches input error

```
// Exception Handling Example
#include <iostream>
#include <string>
#include <limits>
using namespace std;
int main()
{
     int  top, bot, ans;
     bool  done;
     string const errMsg = "Error, invalid input.";
     do {
          try {
               cout << "\nEnter Data:" << endl;
               cout << " Enter dividend: ";
               cin >> top;
               cout << endl;
               cout << " Enter divisor: ";
               cin >> bot;
               cout << endl;
               if (!cin)
                    throw errMsg;
               if (bot == 0)
                    throw bot;
               ans = top / bot;
               cout << " " << top << " / " << bot << " = " << ans <<
endl;
               done = true;
          }

          catch (string msgStr) {
               cout << msgStr << endl;
               cin.clear();
```

```
            cin.ignore(numeric_limits<streamsize>::max(), '\n');
        }

        catch (int x) {
            cout << "Error, division by " << x << endl;
        }

    } while (!done);
    return 0;
}
```

- Templates
    - key functionality -> allows single code base for related functions (of different types)
    - also referred to as parameterized
    - can be used with built-in types or user-defined types
    - template syntax

```
template function definition;template class declaration
```

- Function Templates
    - function templates
        - allows the 'type' to be passed as a parameter
    - compiler will generate the actual code
        - actual code will be based on template (with the specific type included)
        - compiler will **not** generate code for types not used
    - functions headers are typically placed in header file
    - functions implementation also typically placed in header file
    - needed to ensure that the function template is available for the compiler in order to generate the actual code

- Class Templates
    - allows single code base for a set of related functions (i.e., a class)

- Example (without templates)

```
// Function Overloading#include <iostream>#include <iomanip>#include
<string>using namespace std;int getMax(int a, int b)

{    if ( a > b ) {        return a;    }
```

```
else {        return b;    }}

char    getMax(char a, char b)

{    if ( a > b ) {        return a;    } else {        return b;
}}

double  getMax(double a, double b)

{

  if ( a > b ) {        return a;    }

else {        return b;    }}

string  getMax(string a, string b)

{    if ( a > b ) {        return a;    }

else {        return b;    }}

int main()

{    int     xi=42, yi=7, mi;    double  xf=4.5, yf=7.5, mf;    char
xc='e', yc='d', mc;    string  xs="hello", ys="world", ms;    mi =
getMax(xi, yi);    cout << "max (int) is: " << mi << endl;    mf =
getMax(xf, yf);    cout << "max (double) is: " << mf << endl;    mc =
getMax(xc, yc);    cout << "max (char) is: " << mc << endl;    ms =
getMax(xs, ys);    cout << "max (string) is: " << ms << endl;
return 0;}
```

- Example (with templates)

```
//  Templates#include <iostream>#include <iomanip>#include
<string>using namespace std;template <class myType>myType

getMax (myType a, myType b)

{    if ( a > b ) {        return a;    } else {        return b;
}}

int main()

{    int     xi=42, yi=7, mi;

    double  xf=4.5, yf=7.5, mf;    char    xc='e', yc='d', mc;

    string  xs="hello", ys="world", ms;    mi = getMax(xi, yi);

   cout << "max (int) is: " << mi << endl;

    mf = getMax(xf, yf);
```

```
cout << "max (double) is: " << mf << endl;

mc = getMax(xc, yc);

cout << "max (char) is: " << mc << endl;

ms = getMax(xs, ys);

cout << "max (string) is: " << ms << endl;    return 0;}
```

- Templates
  - key functionality -> allows single code base for related functions (of different types)
  - also referred to as parameterized
  - can be used with built-in types or user-defined types
-
  - compiler will generate the actual code
    - actual code will be based on template (with the specific type included)
    - compiler will **not** generate code for types not used
  - functions headers are typically placed in header file
  - functions implementation also typically placed in header file
    - needed to ensure that the function template is available for the compiler in order to generate the actual code

- Recursion
  - previous solutions have been iterative
  - another approach is 'recursion'
  - recursion works very well for a certain set of problems
  - recursion definition
    - a definition in which something is defined in terms of a smaller version of itself
  - recursive definitions must have one (or more) base cases
    - general case must eventually be reducible to the base case
    - base case stops recursion
  - initially recursion can be confusing
    - logically, you can think of a recursive function as having an unlimited number of copies of itself
    - every call to a recursive function – that is every recursive call, has it’s own code and it’s own set of parameters and local variables

            - after completing a recursive call, control returns to the calling routine (which may be the previous call.
                - the current call must complete before control is returned
        - example -> factorial

```
int fact(int num)
{
    if (num == 0)
        return 1;

    return (num * fact(num-1));
}
```

- Direct and Indirect Recursion
    - a function that calls itself is **directly recursive**
    - a function that calls another functions which results in a call to the original function is **indirectly recursive**
    - **infinite recursion** is when the recursion does not terminate
        - poorly designed function
        - invalid base case
    - when the last line of a recursive function is the recursive call, it is referred to as **tail recursion**
        - useful for compiler optimizations

- Example
    - Fibonacci definition
        - fib (n)
            - 0 if n=0
            - 1 if n=2
            - fib(n-1)+fib(n-2) if n>=2
    - Fibonacci series
        - 0 1 1 2 3 5 8 13 21 34 55 ...
    - Fibonacci code

```
int fib(int n)
{
    if (n == 0 !! n==1)
        return n;

    return ( fib(n-1)+fib(n-2) );
}
```

- Review -> Recursion

    - previous solutions have been iterative
    - another approach is 'recursion'
    - recursion works very well for a certain set of problems
    - recursion definition
        - a definition in which something is defined in terms of a smaller version of itself
        - recursive definitions must have
    - one (or more) base cases
        - general case must eventually be reducible to the base case
        - base case stop recursion
- List Processing
    - list (i.e., array) is a collection of values of the same type
        - *note*, the type could be a struct allowing multiple values per element
        - but, still all the same type
    - basic list processing actions
        - search
        - sort
        - insert
        - delete
        - print
- Searching
    - sequential search
        - start at first element and check every element until item is found or at end of list (and item not found)
        - may be quick if item found early in list, may be long it item neat end of list or not found
        - needed to search unsorted data
    - binary search
        - requires sorted data
        - basic idea
            - check middle of list
                - if item found, done
                - if item < middle item, continue search only first half of list
                - if item > middle item, continue searching upper half of list
        - is fast since
            - not searching all elements
            - cutting search size in half repeatedly

- Sorting Overview
    - the sorting problem has attracted a great deal of research
        - electronic data sorting, PhD thesis, Stanford University, 1956
            - bubble sort
        - simple problem definition
        - complex solution for efficient sorting
    - a sorting algorithm is an algorithm that puts elements of a list in a certain order
        - numerical order
        - lexicographical order (e.g., alphabetical order)
    - fundamental problem in computer science
        - sorting is a common algorithmic task with well-understood properties
        - versions of the sort problem are everywhere
            - small examples ->phones, contact lists
            - medium examples ->company employee lists
            - large examples ->large company customer list
                - IRS taxpayer data base
    - many different sorting algorithms
        - ***time space trade-off***
            - in-place (no extra space used), typically slower
            - use additional space, typically faster
        - different classifications, different trade-offs, different applications, etc.
        - big topic in data structures class
            - comparisons
                - a comparison may be costly, particularly if multiple comparisons are required
            - swaps
                - swapping values, possibly large records may be costly
            - recursion
                - many embedded systems limit or dis-allow recursion
            - ***stability***

                - maintain the relative order of records with equal values
            - ***adaptive***
                - can recognize when data is sorted (early exit)
            - sequential or parallel
                - we will focus on sequential sorting
        - method
            - comparison based sorting (traditional)
            - non-comparison based sorting
                - restricts data sets (i.e., integers, limited range)
    - common sort algorithms
        - bubble sort
            - in place, no extra space
            - works on all types
        - insertion sort
            - in place, no extra space
            - works on all types
        - selection sort
            - in place, no extra space
            - works on all types
        - quick sort (recursive)
            - in place, extra space used (on stack)
            - works on all types
        - counting sort
            - extra space required, possibly a lot
            - main use is integers only
    - visualization
        - helps understand how the algorithm works
        - Sort Algorithm Visualization Site (Links to an external site.) (1)
        - Sort Algorithm Visualization Site (Links to an external site.) (2)
- Sort Algorithm -> Bubble Sort
    - Wikipedia Bubble Sort (Links to an external site.)
    - repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order
    - this is repeated until the list is in order
    - basic algorithm

        - **for (int k=0; k < len-1; k++)**
            - **for (int i=0; i < len-k-1; i++)**
                - **if (arr[i] < arr[i+1])**
                    - **swap(arr[i], arr[i+1])**
    - a complete pass through the list with no swaps occurs only when already in order
        - add the ***swapped*** flag
    - generally not very efficient
    - used in CS 135 to help teach looping concepts
    - example
        - **first pass:**
          (5 1 4 2 8) -> (1 5 4 2 8), swap since 5 > 1.(1 5 4 2 8) -> (1 4 5 2 8), swap since 5 > 4(1 4 5 2 8) -> (1 4 2 5 8), swap since 5 > 2(1 4 2 5 8) -> (1 4 2 5 8), no swapsecond pass:(1 4 2 5 8) -> (1 4 2 5 8)(1 4 2 5 8) -> (1 2 4 5 8), swap since 4 > 2(1 2 4 5 8) -> (1 2 4 5 8)(1 2 4 5 8) -> (1 2 4 5 8)third pass:(1 2 4 5 8) -> (1 2 4 5 8)(1 2 4 5 8) -> (1 2 4 5 8)(1 2 4 5 8) -> (1 2 4 5 8)(1 2 4 5 8) -> (1 2 4 5 8)no swaps, early termination
- Sort Algorithm -> Selection Sort
    - Wikipedia Selection Sort (Links to an external site.)
    - works by
        - finding the minimum
        - placing at the start
        - repeat for the rest of the list
    - not adaptive
    - minimizes the number of swaps
    - well suited for applications where the cost of swapping items is high
    - does not recognize when data is sorted
    - not stable sort
    - example:
        - **first pass:**
          (5 1 4 2 8) -> (1 5 4 2 8), swap since 1 is minsecond pass:(1 5 4 2 8) -> (1 2 4 5 8), swap since 2 is minthird pass:(1 2 4 5 8) -> (1 2 4 5 8), swap since 4 is minfourth Pass:(1 2 4 5 8) -> (1 2 4 5 8), swap since 5 is min
- Sort Algorithm -> Counting Sort
    - Wikipedia Counting Sort (Links to an external site.)

    - works by
        - using extra space to store a count of key values
        - space needed based on key size
        - limits data types
            - difficult, but not impossible, to sort floats
        - sort key
            - integer data range
            - example, test scores, 0-100
    - uses extra space, potentially a lot of extra space
    - generally not stable sort
        - can be made stable with extra code
    - not adaptive
        - does not recognize when data is sorted
    - example
        - **data:(5 1 4 2 8 1 5 4 1 8)count array, initial(0 0 0 0 0 0 0 0 0 0)count array(0 3 1 0 2 2 0 0 2 0)data:(1 1 1 2 4 4 5 4 8 8)**
- Sort Algorithm -> Quick Sort
    - Wikipedia Quicksort (Links to an external site.)
    - works by
    - divide and conquer algorithm
        - divides a large array into two smaller sub-arrays
            - the low elements and the high elements
        - quicksort can then recursively sort the sub-arrays
        - the steps are
            - pick an element, called a pivot, from the array
            - reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way)
            - after this partitioning, the pivot is in its final position. This is called the partition operation
                - recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values
        - the base case of the recursion is arrays of size zero or one, which is sorted by definition
        - in practice, is easier to use in-place sort to sort <= 10 elements

- pivot selection
  - pivot selection is important
    - median of three
    - use middle (len/2) element
    - use first or last element
  - poor pivot selection can lead to O(n^2) performance
- complexity
  - run-time
    - best Ω(n log n)
    - typical/expected O(n log n)
    - worst O(n^2)
  - space, O(n), O(n log n) additional space with worst case O(n)
  - not stable sort
- not adaptive
  - does not recognize when data is sorted
- example
  - *note*, this uses pivot as arr[hi]
  - **original array -> call with for entire array (0 to len-1)**
    (81 13 92 43 65 31 57)
  - **step A, selected pivot -> 57**
  - **partition(13 43 31) 57 (65 92 81)**
  - **two sub arrays identified**
  - **(step A)**
  - **call twice**
  - **(for each sub array, excluding pivot)**
  - **first call will be left**
  - **sub arrayfinal array13 31 43 57 65 81 92**

- Vector Type (class)
  - key topic in data structure class
  - C++ provide a vector type to implement a list
  - templated class in library
    - example: **`vector<int> intList;`**
    - example: **`vector<double> realList;`**
  - hides implementation details
    - dynamic allocation, element insertion, element deletion, iteration

    - automatically tracks and maintains length

        - similar to a string
    - iterator
        - loop through all elements in the list or set
        - internal data structure unknown to client program
        - just want to get the next element, which may or may not have a traditional index like a standard array
        - so, can iterate as follows:
            - **for(auto it=begin(v); it != end(v); ++it)**
                - **cout << *it << endl;**
        - don't need to know any implementation details, including the size, just works
- Linked Lists
    - nodes
        - link (pointer)
        - data
    - pointers
        - head
            - typically used to point to the first node (or start of a linked list)
        - current
            - typically used to point to the current node during a traversal
        - node definition
            - typically a struct declaration
            - typically a variable(s) declaration(s)
            - example
            - ```
              struct nodeType
              {
                  int dataItem;
                  nodeType *link;
              }
              ```
            -
- Linked List Types
    - unordered
        - no specific order
    - ordered
        - specific order
        - ascending or descending

- Linked List Operation -> Traversal
    - searching, inserting, and deleting
              ```

    - typically requires accessing or traversing the linked list
    - example
        - assuming previous node declaration
        - sum data item for all nodes
        - output each data item
        - pointer ***head*** points to start of linked list
        - ```
          int sum = 0;
          current = head;
          while (current != NULL) {
              sum += current->info;
              cout << current->info << " ";
              current = current->link;
          }
          ```
        -
- Linked List Operation -> Insertion
    - insertion process depends of linked list type
    - unordered
    - ordered
    - example
        - assuming previous declarations
        - unordered
        - insert at beginning
        - create new node with data item set to 73
        - pointer *head* points to start of linked list
        - example
            - ```
              newNode = new nodeType;
              newNode->info = 73;
              newNode->link = head->link;
              head->link = newNode;
              ```
            -
- Linked List Operation -> Deletion
    - deletion process depends of linked list type
        - unordered
            - must search entire list for item
        - ordered
            - may stop early during search (based on order or linked list)
    - example
        - assuming previous declarations
        - unordered
        - search entire linked list for item
            - stop at last node (link = NULL)

        - delete node with data item set to 42 (if exists)
        - pointer *head* points to start of linked list
        - check cases
            - empty list
            - can not do anything
        - first item is the one to be deleted
            - check if only one node in list
            - must adjust head pointer
        - general case
            - find item
            - if found, remove item
        - item not found
    - exmaple
        - ```
          int removeItem = 42; // item to remove...
          bool found;
          nodeType *prev;
          if (head == NULL) {
              cout << "Can not remove from empty linked list."
          << endll;
          } else {
              if (head->info == removeItem) {
                  current = head;
                  head = head->link;
                  delete current;
              } else {
                  found = false;
                  prev = head; // first node
                  current = head->link; // second node

                  while (current != NULL) {
                      if (current->info != removeItem) {
                          prev = current
                          current = current->link;
                      } else {
                          found = true;
                      }
                  }
                  if (found) {
                      prev->link = current->link;
                      delete current;
                  } else {
                      cout << "Item to be removed not found." <<
          ```
          ```

```
            endl;
                    }
                }
            }
```

- **Operator Overloading**
  - example, operator overloading
    - + (plus) used on integers, returns integers
    - + (plus) used on floats, returns floats
    - + (plus) used on strings, concatenates
  - most existing operators can be overloaded
  - must write header and body
  - use **operator** keyword
  - syntax for used-defined operator overloading
    - <returnType> **operator** operatorSymbol (formal parameter list)
  - overloading limitations
    - can not change precedence
    - can not change associativity
    - default parameters can not be used
    - can not change the number of parameters an operator takes
    - can not create new operators
    - the meaning of how an operator works with default parameters can not be changed
    - operators that can not be overloaded
      - **., .* ::, ?:, sizeof**
  - already familiar with limited operator overloading
    - example, **+**, **-**, ***,** and **/** all work with integers, floats, and doubles
- Why Operator Overloading
  - could work around for objects by including specialized functions.
    - for example, equality
      - if (myClock.isEqual(yourClock)) { // test for equality
    - this is awkward
    - an improved approach would be
      - if (myClock == yourClock)
    - better syntax, easier to read and understand
- Operator Overloading -> Member Functions

- overloading operators as member functions same as other functions
- operators are typically overloaded as member function
  - can be overloaded as friend functions
- requires scope resolution operator, ::, and keyword ***operator***
- example:
  - overload == (equal) operator for complex numbers as member function
  - overload != (not equal) operator for complex numbers as member function

HEADER FILE:

```
bool complexType::operator != (const complexType&);
bool complexType::operator == (const complexType&);
```

IMPLEMENTATION FILE:

```
bool complexType::operator != (const complexType& two) {
      return(realPart != two.realPart &&
     imaginaryPart != two.imaginaryPart);
}

bool complexType::operator == (const complexType& two) {
      return(realPart == two.realPart &&
     imaginaryPart == two.imaginaryPart);
}
```

- Friend Functions
  - a friend function is a non-member function of the class
  - has full access to all class members (public, protected, and private)
  - requires **friend** keyword in header file
  - most functions and operator overloading can be either member or non-member
  - typical use cases of friend functions are operations that are conducted between two different classes accessing private or protected members of both
    - friend functions are rarely needed and rarely used
  - when overloading the stream operators, they must be overloaded as friend functions

        - overloading the stream operators will not work as member functions
- Operator Overloading -> non-Member Functions
    - example:
    - overload != (not equal) operator for complex numbers as friend function
    - overload == (not equal) operator for complex numbers as friend function
- Overloading + Operator
    - typically overloaded as member function
        - can be overloaded as friend functions
    - function prototype
        - complexType operator + (const complexType&);
    - operands
        - result = operand1 + operand2
        - operand 1 -> class variables for current object
        - operand2 parameter must be reference
            - typically const since must not alter input values
    - example:

```
HEADER FILE:
complexType operator + (const complexType&);

IMPLEMENTATION FILE:
complexType complexType::operator + (const complexType& one) {
      complexType temp;

      temp.realPart = realPart + one.realPart;
      temp.imaginaryPart = imaginaryPart + one.imaginaryPart;

      return temp;
}
```

- Overloading << and >> operators
    - must be overloaded as friend functions
    - function prototype
        - ostream operator << (ostream&, const className&);
    - both parameters must be reference parameters
    - for <<

        - the first parameter must reference ostream (output stream)
        - the second parameter, usually a const reference to the class
        - the function return type is a reference to an ostream object
    - for >>
        - the first parameter must reference istream (input stream)
        - the second parameter, is a reference to the class
        - the function return type is a reference to an istream object
    - example:

```
HEADER FILE:
friend ostream& operator<< (ostream&, const complexType&);

IMPLEMENTATION FILE:
ostream& operator << (ostream& os, const complexType& cNum) {
      os << "(" << cNum.realPart << ", " << cNum.imaginaryPart << ")";
      return os;
}
```

- Pointer **this**
    - a member function can directly access class variables
    - sometimes a member function can refers to the object as a whole
    - every object contains a hidden pointer to itself (this)

- **Templates**
    - key functionality -> allows single code base for related functions (of different types)
    - also referred to as parameterized
    - can be used with built-in types or user-defined types
    - template syntax

```
template <class myType>
function definition;

template <class myType>
class declaration
```

- Function Templates
  - function templates
  - allows the 'type' to be passed as a parameter
  - compiler will generate the actual code
  - actual code will be based on template (with the specific type included)
  - compiler will **not** generate code for types not used
  - functions headers are typically placed in header file
  - functions implementation also typically placed in header file
  - needed to ensure that the function template is available for the compiler in order to generate the actual code
- Class Templates
  - allows single code base for a set of related functions (i.e., a class)
- Example (with templates)

```
// Templates
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

template <class myType>
myType getMax (myType a, myType b)
{
     if ( a > b ) {
          return  a;
     } else {
          return b;
     }
}

int main()
{
     int xi=42, yi=7, mi;
     double xf=4.5, yf=7.5, mf;
     char xc='e', yc='d', mc;
     string xs="hello", ys="world", ms;

     mi = getMax<int>(xi, yi);
     cout << "max (int) is: " << mi << endl;
```

```
    mf = getMax<double>(xf, yf);
    cout << "max (double) is: " << mf << endl;

    mc = getMax<char>(xc, yc);
    cout << "max (char) is: " << mc << endl;

    ms = getMax<string>(xs, ys);
    cout << "max (string) is: " << ms << endl;

    return 0;
}
```

-
    - allows single code base for different types

- **Exceptions**
    - undesired situation during program execution
    - need a clean method to handle exceptions
- Try/Catch Block
    - a try/catch block allows exceptions to be trapped
        - avoid run-time crashes
        - allows programmer chance to address exception
        - programmer must address or clear exception cause
    - do statements in **try block**
        - code is placed in a try { } block
    - if no exception is generated
        - no problem :-)
        - none of the catch blocks are executed
    - if exception is generated
        - program does not crash
        - catch block code is executed
    - catch blocks
        - exception handlers are in catch { } blocks
        - catch block syntax: catch (<type> <var>)
        - the order of the catches can be important

        - ellipses (...) are used to catch any type
            - catches any other non-specified exception
            - *note*, not used in this small example code
    - throw statement
        - exceptions must be thrown to a catch block
        - syntax: throw <expression>
        - the type of the expression is used to select the specific catch block
    - general try/catch syntax summary

- **Recursion**
    - previous solutions have been iterative
    - another approach is 'recursion'
    - recursion works very well for a certain set of problems
    - recursion definition
        - a definition in which something is defined in terms of a smaller version of itself
    - recursive definitions must have one (or more) base cases
        - general case must eventually be reducible to the base case
        - base case stops recursion
    - initially recursion can be confusing
        - logically, you can think of a recursive function as having an unlimited number of copies of itself
        - every call to a recursive function – that is every recursive call, has it's own code and it's own set of parameters and local variables
        - after completing a recursive call, control returns to the calling routine (which may be the previous call.
            - the current call must complete before control is returned
- Direct and Indirect Recursion
    - a function that calls itself is **directly recursive**
    - a function that calls another functions which results in a call to the original function is **indirectly recursive**
    - **infinite recursion** is when the recursion does not terminate
        - poorly designed function
        - invalid base case

    - when the last line of a recursive function is the recursive call, it is referred to as **tail recursion**
        - useful for compiler optimizations

- **List Processing**
    - List (i.e., array) is a collection of values of the same type
        - *note*, the type could be a struct allowing multiple values per element
        - but, still all the same type
    - Basic list processing actions
        - search
        - sort
        - insert
        - delete
        - print
- Searching
    - sequential search
        - start at first element and check every element until item is found or at end of list (and item not found)
        - may be quick if item found early in list, may be long it item neat end of list or not found
        - needed to search unsorted data
    - binary search
        - requires sorted data
        - basic idea
            - check middle of list
                - if item found, done
                - if item < middle item, continue search only first half of list
                - if item > middle item, continue searching upper half of list
        - is fast since
            - not searching all elements
            - cutting search size in half repeatedly
- Sorting
    - Fundamental problem in computer science
        - versions of the sort problem are everywhere

            - small examples
                - phones, contact lists

        - medium examples
            - company employee lists
        - large examples
            - large company customer list
            - IRS taxpayer data base
    - Many different sorting algorithms
        - different trade-offs, different applications, etc.
            - big topic in data structure class
        - time space tradoff
            - in-place (no extra space used), typically slower
            - use additional space, typically faster
        - comparisons
            - a comparison may be costly, particularly if multiple requires are required
        - swaps
            - swapping values, possibly large records may be costly
    - Common sort algorithms
        - bubble sort
            - in place, no extra space
            - works on all types
        - insertion sort
            - in place, no extra space
            - works on all types
        - selection sort
            - in place, no extra space
            - works on all types
        - quick sort (recursive)
            - in place, extra space used (on stack)
            - works on all types
        - counting sort
            - extra space required, possibly a lot
            - main use is integers only

Lecture Topics

- Review -> Linked Lists
  - nodes
    - link (pointer)
    - data
  - pointers
    - head
      - typically used to point to the first node (or start of a linked list)
    - current
      - typically used to point to the current node during a traversal
    - node definition
      - typically a struct declaration
      - typically a variable(s) declaration(s)

- Linked List Types
  - unordered
    - no specific order
  - ordered
    - specific order
    - ascending or descending

- Linked List Operation -> Traversal / Searching
  - searching and/or printing
  - requires accessing or traversing the linked list from the beginning
  - search process depends of linked list type
    - unordered
      - must search entire list for item
    - ordered
      - may stop early during search (based on order or linked list)
  - once location of insertion is determined must
    - create and populate new node
    - updated link pointers to insert

- Linked List Operation -> Insertion
  - insertion process depends of linked list type
    - unordered

            - insert at either end
        - ordered
            - traverse until correct ordered location is found

- Linked List Operation -> Deletion (of single item)
    - deletion process depends of linked list type
        - unordered
            - must search entire list for item to delete
        - ordered
            - may stop early during search (based on order or linked list)
    - once found must
        - remove from linked list by updated link pointers
        - delete node

- Linked List -> Stack
    - implementation, unordered linked list
    - LIFO, last-in, first-out
    - insertion
        - always at beginning of list
    - removal
        - always from beginning of list

- Linked List -> Queue
    - implementation, unordered linked list
    - FIFO, first-in, first-out
    - insertion
        - always at end of list
    - removal
        - always from beginning of list

```
bool isPalindrome(string str)

{

linkedStacked<char> stk;

int len = str.length();

for( int i = 0; i <len; i++)

{stk.push(str[i]);}
```

```
for(int i = 0;  i < len; i++)
{stk.push(str[i]);}

for(int i =0; i<len; i++)
{ if(stk.top()!=str[i])
{return false}
stk.pop();
}
return true;
}
```

```
bool isPalindrome(string str)
{
linkedQueue<char> q;
int len = str.length();
for( int i = 0; i <len; i++)
{q.enqueue(str[i]);}


for(int i =0; i<len; i++)
{ if(q.front()!=str[i])
{return false}
q.dequeue();
}

return true;
}
```