Inheritance

- A relationship between classes
- One class inherits from another
    - can inherit from multiple classes
- **an "is a" relationship**

A **subclass** is either:
- a more specific version of its parent (superclass)
- beware an enhancement of its superclass

Inherit-from	Inherits
superclass	subclass
base	derived
Base	descendant

A **protected** member of a class is accessible to class member *& those of its descendants*

When a : is used to invoke a **superclass**, the **superclass** constructor is used completely

Access Right of Derived Classes

	private	protected	public
private	.	.	.
protect	private	protected	protected
public	private	protected	public

Static functions are not part of the class, there is one copy, which everyone shares. It is not inherited.

A *virtual* function in C++ is a base class member function that you can redefine in a derived class to achieve polymorphism. You can declare the function in the base class using the *virtual* keyword

TheDate, a pointer in the DaysElapsedInYear example, will call the daysElapsed() function, according to whether it is pointing at a Year object, or a LeapYear object. This is true when functions are overwritten in class hierarchies, IF the function that appear in multiple class is designated as *virtual*. The keyword *virtual* switches ON polymorphism.

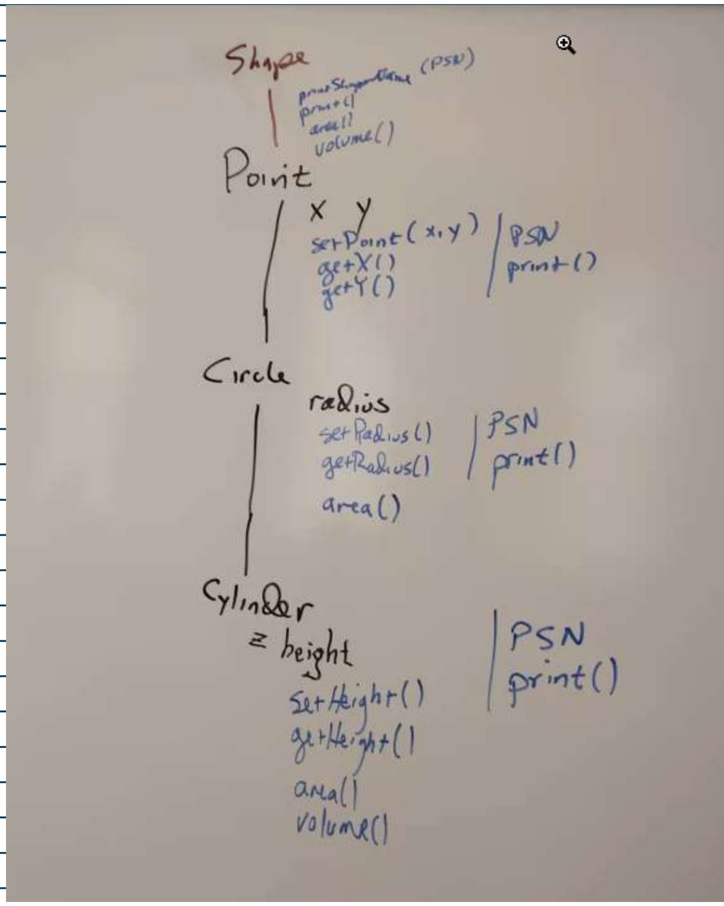
Without the *virtual* keyword in daysElapsed(), then the call of daysElapsed will be based on the type of the pointer, not what it points at. E.g. If we have Year *TheDate and daysElapsed is NOT *virtual*, then whether daysElapsed points at a Year object of a LeapYear, the version of daysElapsed in the Year class will be called. A pure *virtual* function, once inherited, must be implemented for a subclass to be instantiable.

Polymorphism is the ability in a program to call one of multiple versions of a function, overwritten in one or more subclasses, according to the disposition of a pointer. I.e. It will call the particular version of the function designated *virtual* based on which member of the class hierarchy a pointer is pointed at

An *abstract class*, has at least one function that is prototyped, but is not implemented. Because of this, it CANNOT be instantiated. These function will be declared as pure *virtual*. A pure *virtual* function, is one that has its prototype <u>set to 0</u>. It won't, and can't be implemented in the class, but it MUST be implemented in subclasses so that they can be instantiated. A pointer to an *abstract class*, can be used to instantiate that class, but can still be used to point to and establish non-abstract classes lower in the class hierarchy

Inheritance 2

Gardi@kutztown.edu



The keyword *virtual* turns on **dynamic binding**, which means function calls are hooked up to function definitions at run-time, instead of the linker doing this binding.

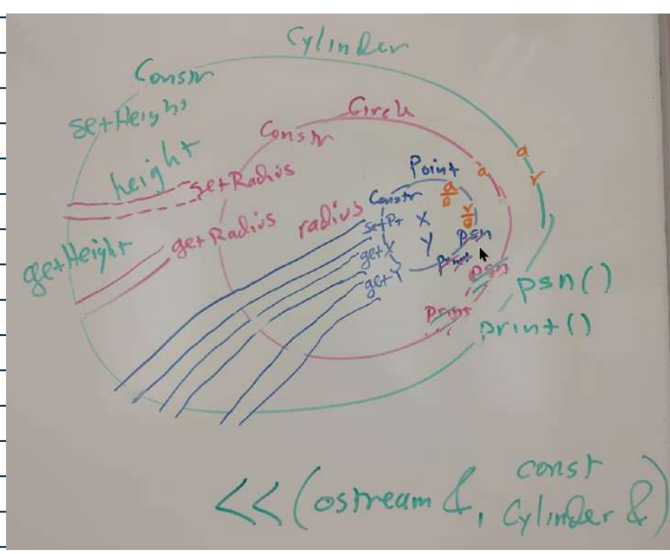
If there is a pointer to a Circle, cPtr, without dynamic binding, a call of area() would be hooked up to the area function declared in Circle by the linker.

With dynamic binding, the call isn't hooked up by the linker. The function to be run will be dependent upon where the Circle pointer is pointing. If it is pointing at a Cylinder, then Cylinder's version of area() will be called.

A class can only be instantiated if it has no pure virtual member functions.
- For example, Shape has two pure virtual functions.
  * Point is going to implement them and therefore Point can be instantiated.

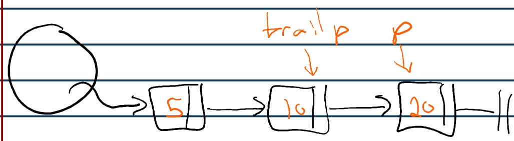
Review: A pointer to Shape can be declared, but can't be pointed at a Shape.
- A Shape is not instantiable (it has pure virtual functions)
- The pointer can be pointed at instantiable Shape subclasses.

Shape Hierarchy
- Every descendant will have area() and volume(), whether inherited or overwritten.
- At least one descendant must implement printShapeName() (psn) and print() for this to have any use



Variations on Linked List

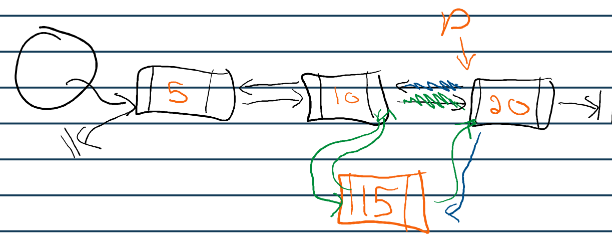
Singly-linked List



insert 15:

trailp->next = new node(15,p);

Doubly-linked List

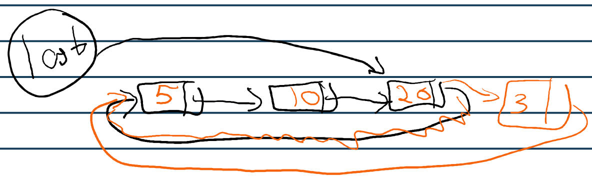


insert 15:

p->prev->next = new node(15,p->prev,p);

p->prev = p->prev->next;

Circular Linked List



- constant time insert at end AND beginning
- main pointer points at the last node

insertAtBeginning:

last->next = new node(,last->next);

L-value R-Value

An L-value is a changeable memory reference
An R-value is a read-only memory reference

Analysis of Algorithms

Time Complexity
- exact count of operations T(n) as a function of input size *n*
- complexity analysis using O(..) bounds
- constant time, linear, logarithmic, exponential, … complexity

Complexity analysis of basic data structures' operations.
Linear and *Binary Search* algorithms and their analysis.
Basic Sorting algorithms and their analysis

Efficiency of an algorithm can be measured in terms of:
- execution time (*time complexity*)
- memory required (*space complexity*)

Factors that ***should not*** affect time complexity analysis:
- the programming language chosen
- the quality of the complier
- the speed of the computer on which the algorithm is running

Simplified analysis can be based upon:
- number of arithmetic operations or comparisons performed
- number of times through a critical loop
- number of array elements accessed, etc.

Polynomial evaluation
Brute force method: p(x) = 4*x*x*x*x + 7*x*x*x - 2*x*x + 3*x +6
Horner's method: p(x) = (((4*x+7)*x - 2)*x + 3)*x + 6)

Horner's method is more efficient, since fewer operations O(n) vs O(n^2)

Search Algorithms

The problem: Search an array *a* size of *n* to determine whether the array contains the value *key*; return index if found, -1 if not found

Linear Search

```
Set k to 0.
While (k < n) and (a[k] is not key)
    Add 1 to k
If(key is a[i]) return k
return -1
```

This algorithm is *O(n)*

Binary Search

```
Set m to 0
Set k to floor((n-m)/2)
While (m < n) and (a[k] is not key)
    If(a[k] < key) m = k + 1
    Else n = k - 1
    k = floor((n-m)/2)
If(key is a[i] return k
return -1
```

This algorithm is *O(Log n)*

Sort Algorithms

Selection Sort

- list with *n* elements
- must place *n-1* elements
- main operation is a comparison

1st time: compare *n* elements
2nd time: compare *n-1* elements
3rd time: compare *n-2* elements
. . .
n-1st time: compare *2* elements

O(n^2)

Recursion & Recurrence

Recurrence is when the next value is a function of one or more previous values. The goal is, if possible, to find a closed form.

- one of the most popular recurrence relations is n!
    - 0! = 1
    - n! = n * (n-1)!

Expansion Method for Solving Recurrence Relations

- write out recursive definition
- expand recursive terms using their definitions
- continue until a pattern is observed
- once a pattern is found, put into summation notation
- achieve a function for recurrence relation
- double check using a inductive proof