Abstract Data Types (ADTs)

- A functional approach to describing information storage and access.
  - No standardized approach.
  - Not directly supported by language
  - Requires discipline by programmer

Object Oriented Programming (OOP)

- Three Precepts
  - Inheritance
  - Polymorphism
  - Encapsulation
- Not a goal of this course

OOP: Polymorphism

Java supports pseudo-polymorphism

- e.g. operator overloading
- e.g. templates

OOP: Encapsulation

- ***Information hiding***
- ***Access control***
- Supported in C++ via classes.
- Supported in Java via classes, interfaces and packages
- Public versus private declarations control access

Data Abstraction

Data abstraction requires us to divorce

1) data and how it is stored

from

2) the way it is used

Data Abstraction

- User of data does not need to know
  - How it is stored or
  - How it is organized
- The user only concerned with
  - **How can information be used**
  - **How can I access data**

ADTs

- ADT is **A**bstract **D**ata **T**ype
- An ADT is a Black Box
  – We can't see what's inside (implementation)
  – We only see what goes in and comes out. (interface)

ADTs

In this course we will wear two hats

1)User hat

- we are outside the black box

2) Programmer hat

- we are inside the black box

ADTs: Example

Consider a clock.

- Single function - return time of day
- Lot of implementations:
  – Battery operated,
  – 60 cycle AC
  – Pendulum,
  – Water driven

ADTs: Example

- Some implementations impose restrictions.
  - Pendulum has more space requirements
  - AC power requires electrical source
- Some implementations are better choice
- Situationally dependent

ADTs

1)User hat
  - Only consider the functionality
  - How is information used/ accessed

2) Programmer hat (Later)
  - Consider inmplementation tradeoffs
  - Consider how it is stored/organized

ADT Example

- Problem:

  Design a new model of automobile

- Designer - outside the black box
- Engineer - inside the black box

Designer

- Thinks about how vehicle will look
  - Set a trend?
  - Be part of the crowd?
- How will vehicle appeal to a particular market segment

Engineer

- Thinks about implementation
- How will the parts fit together?
- Can existing factory tooling be used for all or part of manufacture?
- Costs?

Example: engine

- Designer identifies requirements/priorities
- Fuel efficiency/performance
- Size or weight restrictions
- Must be internal combustion engine
  - Existing societal infrastructure
  - Eliminates battery/electric power options

Example: engine

- Engineer Identifies implementations and necessary tradeoffs
- Piston-based engine
  - Complicated
  - Gasoline powered
  - Diesel powered
- Wankel rotary engine

Example: engine - Gasoline

- Range of fuel economy
- Better performance
- Less pollution than diesel
- Improve fuel efficiency with alcohol mixes

Example: engine - Diesel

- Good fuel economy
- Strong low end torque
- Poor acceleration
- Difficult to start in cold weather
- Fuel is less readily available

Example: engine - Wankel

- Simple, easy to maintain
- Poor fuel economy
- Turbo charger
  - Improves the fuel economy
  - Increases complexity

ADTs: Specifying an ADT

- Good method is to write as a class.
- Use preconditions, postconditions, invariants
- No function bodies needed.

ADT Format

ADT *Name* **is**
Data
Describe the nature of the data and any initialization.
Methods
Method$_1$
Input: Data from the client.
Precondition: Necessary state of the system (what needs to be true) before executing the operation
Process: Actions performed with the data.
Postcondition: State of the system (what needs to be true) after executing the operation.
Output: Data values returned to the client.

Method$_2$ ...
Method$_n$ ...
end ADT *Name*

ADT Example: Dice

ADT Dice

Data

A count, N, of the number of dice in a single toss, the sum of the toss, and a list of the N tossed die values. Values of a toss range from 1 to 6. Sum ranges from 1N to 6N.

Methods

(next slide)

Methods

Toss

Input:	None
Precondition:	None
Process:	Toss the dice and compute the sum.
Postcondition:	The sum contains the sum of the dice on the toss, and the list identifies the value of each die in the toss.
Output:	None

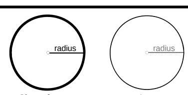
DieTotal

Input:	None
Precondition:	None
Process:	Retrieve the value of the variable which specifies the sum for the most recent toss.
Postcondition:	None
Output:	Return the total of the dice for the most recent toss.

DisplayToss

Input:	None
Precondition:	None
Process:	Print the list of dice values for the most recent toss.
Output:	None
Postcondition:	None

End ADT *Dice*



ADT Circle **is**

Data

A non-negative real number specifying the radius of the circle, initialized to a non zero real radius.

Methods

Area

Input:	None
Precondition:	None
Process:	Compute the area of the circle.
Postcondition:	None
Output:	Return the area.

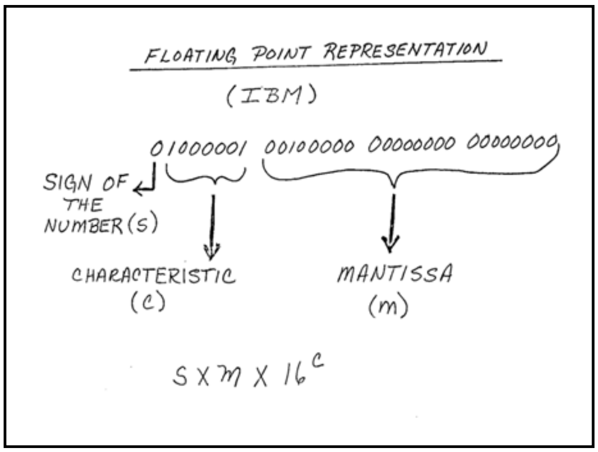
Circumference

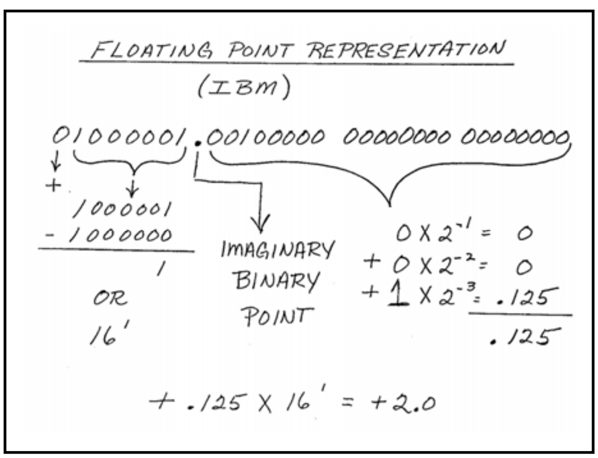
Input:	None
Precondition:	None
Process:	Compute the circumference of the circle.
Postcondition:	None
Output:	Return the circumference.

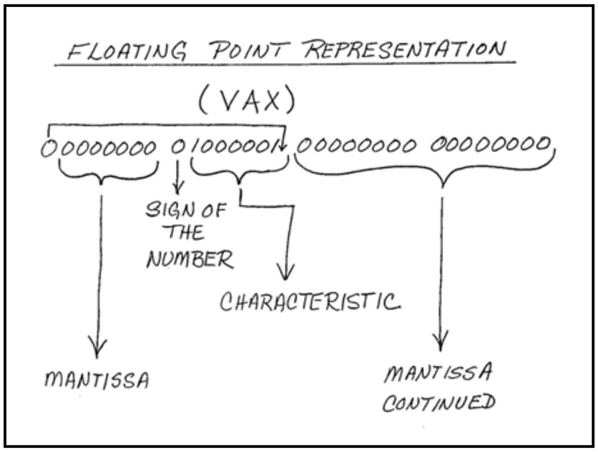
end ADT Circle

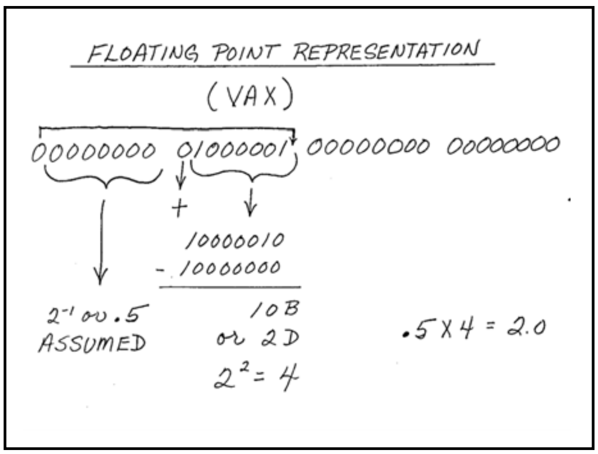
```
#include <iostream.h>
const float PI = 3.14152;
//declare Circle class with data and method declarations
class Circle
{
        private:
                float   radius;         //initialize to a positive value

        public:
                Circle (float r);       //constructor
                float Circumference(void) const;
                float Area(void) const;
};
```

Note structural similarity between ADT and Class declaration






FLOATING POINT REPRESENTATION
(VAX)
00000000 01000001 00000000 00000000
SIGN OF THE NUMBER
CHARACTERISTIC
MANTISSA
MANTISSA CONTINUED


FLOATING POINT REPRESENTATION
(VAX)
00000000 01000001 00000000 00000000
+
10000010
- 10000000
10B
or 2D
2² = 4
2⁻¹ or .5
ASSUMED
.5 X 4 = 2.0