ABSTRACT DATA TYPES

Problem Solving with Computers-II

Today's goals

- Defining Abstract Data Types
- Different ways of initializing objects and when to use each:
  - Default constructor
  - Parametrized constructor
  - Parameterized constructor with default value
- Operator overloading as an example of compile time polymorphism
  - what is operator overloading?
  - why/when would we need to overload operators?
  - how to overload operators in C++ ?
- Linked List
  - Procedural implementation vs OOP style
  - Using recursion to implement linked list operations

Abstract Data Type (ADT)

- Abstract Data Type (ADT) is defined by data + operations on the data.
- Key features
  - **Abstraction:** hide implementation details
  - **Encapsulation:** bundle data and operations on the data, restrict access to data only through permitted operations

```
class IntList {
public:
    IntList();
    // other public methods


private:
    struct Node {
        int info;
        Node* next;
    };
    Node* head;
    Node* tail;
};
```

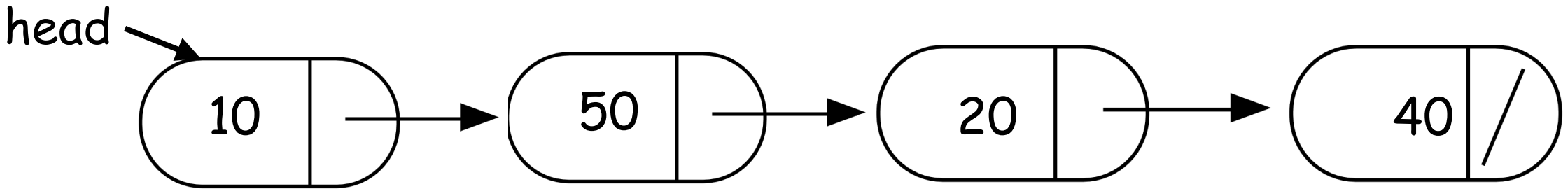
Questions to ask about any ADT:

- **What operations does the ADT support?**

  The list ADT supports the following operations on a sequence:

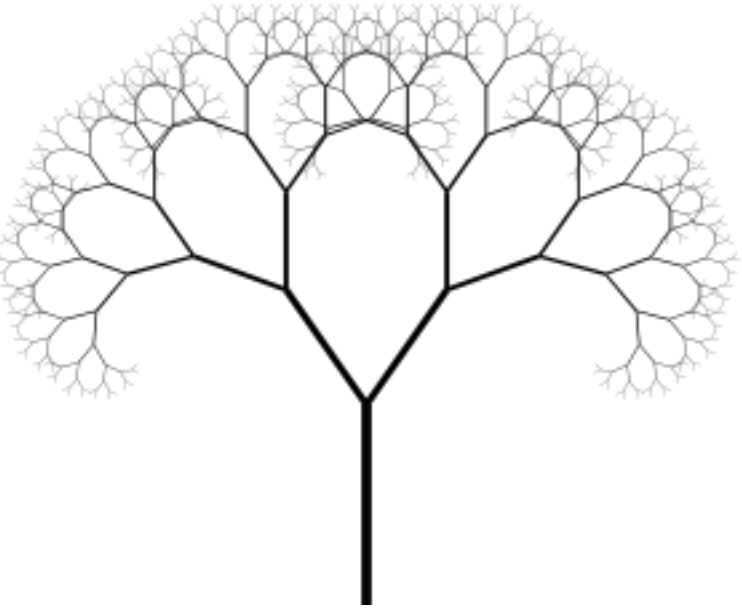
  1. push_front (add a value to the beginning of the sequence)
  2. push_back (add a value to the end of the sequence)
  3. pop_front (delete the first value in the sequence)
  4. pop_back (delete the last value in he sequence)
  5. front() (return the first value)
  6. back() (return the last value)
  7. delete (a value)
  8. print all values

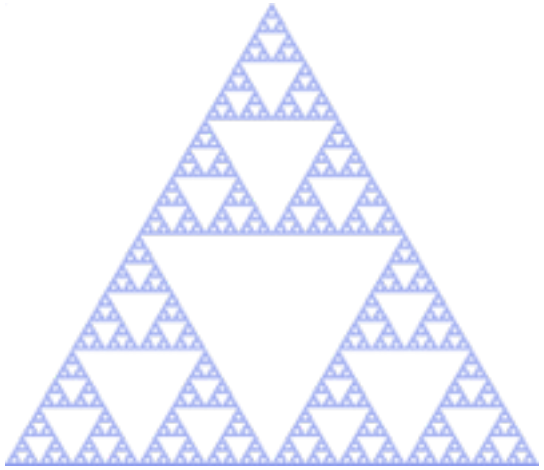
- **How do you implement each operation (data structure used)?**
- **How fast is each operation?**




```
int IntList::push_front(int value){

     //add value to the beginning of the sequence
  }
```

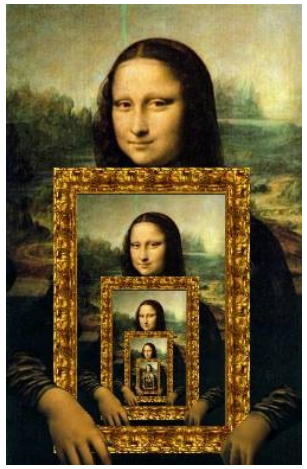
Recursion



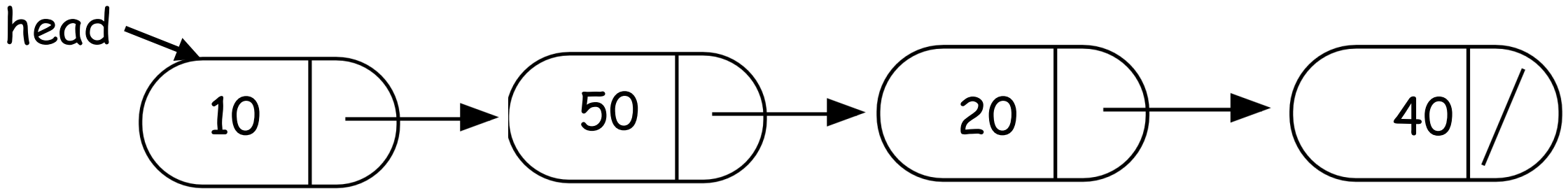


Sierpinski triangle

Zooming into a Koch's snowflake



Using recursion to implement operators involving a linked list



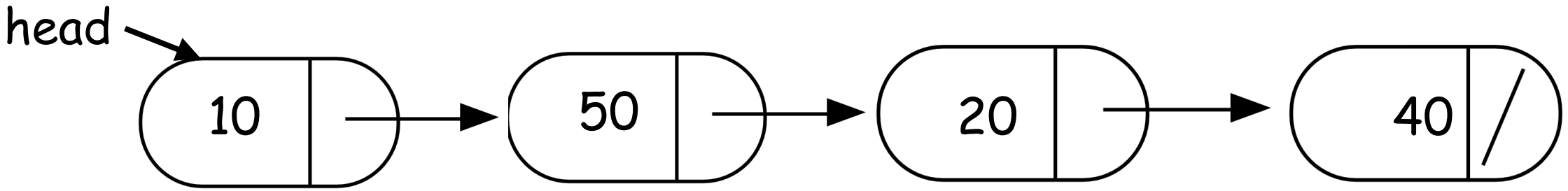

```
int IntList::sum(){

    //return the sum of the sequence
}
```

Helper functions

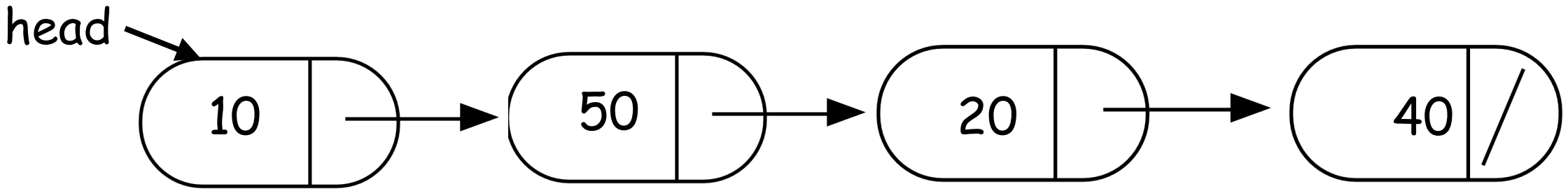
- Sometimes your functions takes an input that is not easy to recurse on
- In that case define a new function with appropriate parameters: This is your helper function
- Call the helper function to perform the recursion
- Usually the helper function is private

For example

```
Int IntList::sum(){

   return sum(head);
    //helper function that performs the recursion.

}
```




```
int IntList::sum(Node* p){




}
```




```
bool IntList::clear(Node* p){









  }
```

Approximate Terminology

- instance = object
- field = instance variable
- method = function
- sending a message to an object = calling a function

Will this code compile?

```
int main(){
  Complex p;
  Complex w(1, 2);
  p = w;
  p.conjugate();
  p.print();
}
```

A. Yes
B. No
C. I am not sure . . .

```
class Complex
{
private:
    double real;
    double imag;
public:
    double getMagnitude() const;
    double getReal() const;
    double getImaginary() const;
    void print() const;
    void conjugate();
    void setReal(double r);
    void setImag(double r);
};
```

Will this code compile?

```
int main(){
  Complex p;
  Complex w(1, 2);
  p = w;
  p.conjugate();
  p.print();
}
```

A. **Yes**
B. **No: We need a parametrized constructor**
C. **I am not sure . . .**

```
class Complex
{
private:
    double real;
    double imag;
public:
    Complex(double re = 0, double im = 0);
    double getMagnitude() const;
    double getReal() const;
    double getImaginary() const;
    void print() const;
    void conjugate();
    void setReal(double r);
    void setImag(double r);
};
```

Polymorphism: same code different behavior

Example: overloading functions, operator overloading

Overloading the + operator for Complex objects

```
p = q + w;
```

Goal: We want to apply the + operator to Complex type objects

New method: add()

```
int main(){
  Complex p;
  Complex q(2, 3);
  Complex w(10, -5);
  w.conjugate();
  p = add(q, w);
  p.print();
}
```

Approach 1

```
int main(){
  Complex p;
  Complex q(2, 3);
  Complex w(10, -5);
  w.conjugate();
  p = q.add(w);
  p.print()
}
```

Approach 2

Overloading the + operator for Complex objects

```
p = add(q, w);
```

```
p = q.add(w);
```

```
p = q + w;
```

Goal: We want to apply the + operator to Complex type objects

Overloading the << operator

```
int main(){
  Complex w(10, -5);
  w.conjugate();
  w.print();
}
```

```
int main(){
  Complex w(10, -5);
  w.conjugate();
  cout << w;
}
```

```
Before overloading the << operator
```

```
After overloading the << operator
```

```
cout << w;
```

Select any equivalent C++ statement:

```
w.operator<<(cout);
```

A

```
cout.operator<<(w);
```

B

```
operator<<(cout, w);
```

C

```
operator<<(cout, w);
```

Select the function declaration that does NOT match the above call

A
```
void operator<<(ostream &out,
                const Complex &c);
```

B
```
void Complex::operator<<(ostream &out);
```

C
```
Complex operator<<(ostream &out,
                   Complex c);
```

Operator Overloading

We would like to be able to perform operations on two objects of the class using the following operators:

<<

==

!=

+

-

and possibly others

Overloading Operators for IntList

In lab02 you will overload operators for the IntList ADT

==

!=

+ (list concatenation)

<< (overloaded stream operation to print the sequence)

Some advice on designing classes

- Always, *always* strive for a narrow interface
- Follow the principle of abstraction and encapsulation:
  - the caller should know as little as possible about how the method does its job
  - the method should know little or nothing about where or why it is being called
  - Your class is responsible for it's own data; don't allow other classes to easily modify it! Make as much as possible private