OBJECT ORIENTED PROGRAMMING

OBJECT ORIENTED PROGRAMMING

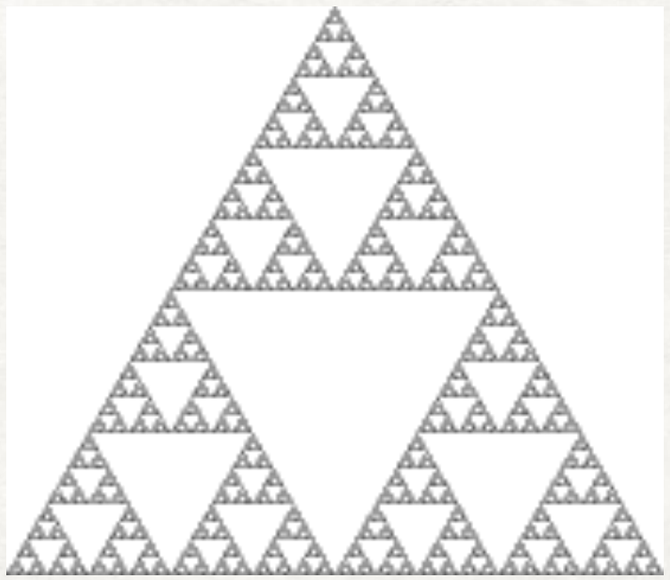
- A programming “paradigm” introduced to overcome limitations
- Sapir-Whorf Hypothesis: Language in which idea is expressed directs that nature of thought
- Characteristics of OOP:
    - Abstraction
    - Encapsulation
    - Inheritance
    - Polymorphism (Overloading)

IMPERATIVE PROGRAMMING

- The “traditional” model of computation
- Describe list of instructions for computer to execute
- Procedural Programming: Division into functions
- **Functions have unrestricted access to global data**
- **Unrelated functions and data are a poor model of the real world**
- Complex real world objects have *attributes* and *behaviour*

OBJECT ORIENTED PROGRAMMING

- See the whole world in the form of objects while designing modules
- Alan Kay: Why construct whole out of pieces that are useless by themselves? Build whole out of pieces that are similar at all levels
- Recursive Design: Build program out of little computing agents



ABSTRACTION & ENCAPSULATION

- Hide background details and provide essential information
- Bundling data and functions together and hiding them

```
#include<iostream>

class Adder {
  public:
    Adder(int i=0) {total=i;}            //Constructor
    void addNum(int num) {total+=num;}  //Interface to outside world
    int getTotal() {return total;}

  private:
    int total;  //Hidden data to outside world
};

void main() {
  Adder a;    //Object of class adder
  a.addNum(10); a.addNum(20);
  cout<<a.getTotal()<<endl;
}
```

INHERITANCE

```
#include<iostream>

class Shape {
  public:
    int width, height;
    void setWidth(int w){width=w;}
    void setHeight(int h){height=h;}
};

class Rectangle {
  public:
    int width, height;
    void setWidth(int w){width=w;}
    void setHeight(int h){height=h;}
    int getArea() {return width*height;}
};
```

```
#include<iostream>

class Shape {
  public:
    int width, height;
    void setWidth(int w){width=w;}
    void setHeight(int h){height=h;}
};

class Rectangle : public Shape{
  public:
    int getArea() {return width*height;}
};
```

- class derivedClass : <access-specifier> baseClass
- Helps in code reusability and faster implementation

ACCESS SPECIFIERS

Members

- public: Accessible from anywhere
- private: Cannot be accessed from outside the class
- protected: Can be accessed in *derived classes*

Inheritance

- public: Only public and protected members of base class can be accessed in derived class
- protected: public and protected members of base class become protected members of derived class
- private (default): public and protected members of base class become private members of derived class

OVERLOADING

```
#include<iostream>

class Complex {
  public:
    Complex(double re, double im):real(re),imag(im){};
  private:
    double real, imag;
};
```

```
void main() {
  Complex c1(1,2);
  Complex c2(3,4);

  //Without op overloading
  //Complex c3 = c1.Add(c2);

  //Op Overloading
  //Complex c3 = c1 + c2;
}
```

```
Complex Complex::operator+(const Complex& other) {
  double res_real = real + other.real;
  double res_imag = imag + other.imag;
  return Complex(res_real, res_imag);
}

void main() {
  Complex c1(1,2);
  Complex c2(3,4);
  Complex c3 = c1 + c2; //Operator overloading
}
```

- Precedence, associativity, arity cannot be changed, cannot redefine meaning of a procedure

FUNCTION OVERLOADING

```
#include<iostream>

class Addition {
  public:
    int sum(int a, int b) {return a+b;}
    int sum(int a, int b, int c) {return a+b+c;}
};


void main() {
  Addition a;
  cout<<a.sum(10,20);
  cout<<a.sum(10,20,30);
}
```

- The definition of the function must differ from each other by types and/or the number of arguments

POLYMORPHISM

- Code/operations/objects behave differently in different contexts
- Function overriding: Hierarchy of classes, related by inheritance

```
class Base {
  public:
    void show() {cout<<“Base class”;}
};

class Derived:public Base {
  public:
    void show() {cout<<“Derived class”;}
};
```

```
void main() {
  Base b;
  Derived d;
  b.show();
  d.show();
}
```

```
Base class
Derived class
```

```
void main() {
  Base *b;
  Derived d;
  b = &d;
  b->show();
}
```

```
Base class
```

- Early binding or static resolution: function show() is set during compilation of the program

VIRTUAL FUNCTIONS

```
class Base {
  public:
    virtual void show() {cout<<“Base class”;}
};

class Derived:public Base {
  public:
    void show() {cout<<“Derived class”;}
};
```

```
void main() {
  Base *b;
  Derived d;
  b = &d;
  b->show();
}
```

Derived class

- Dynamic linkage or late binding: function to be called is selected based on the kind of object for which it is called
- Pure virtual function: No meaningful definition of function in base class

```
virtual void show() = 0; //pure virtual function
```

SUMMARY

- OOP is a new way of *thinking*, not just addition of new features
- Program consists of *objects* which are an encapsulation of *attributes* and *behaviour*
- Behaviour of an object is dictated by its *class*
- Inheritance creates class hierarchies which facilitates code reusability
- Polymorphism: overloading and overriding are powerful concepts

REFERENCES

- Wikipedia
- tutorialspoint.com for C++
- http://www.ent.mrt.ac.lk/~ekulasek/c++/lecture7.pdf
- http://web.engr.oregonstate.edu/~budd/Books/oopintro3e/info