Knowledge Update Course
for Secondary School Computer and IT Teachers

Computer Programming

Day Three

Jasper Wong
email: icjwong@polyu.edu.hk

Industrial Centre
The Hong Kong Polytechnic University

July, 2003

Day Three Agenda

- Structured and Object Oriented Programming
- Basic Concepts of Object-Oriented Programming
  - Objects, Data abstraction and Encapsulation
  - Inheritance, Polymorphism
  - Dynamic binding, Message passing
- Object-Oriented Programming
- Software Engineering
  - Basic Concepts
  - Waterfall Model
  - Fountain Model
  - Object-Oriented Analysis
  - Prototyping Models

Traditional Structured Approach

Object-Oriented Approach

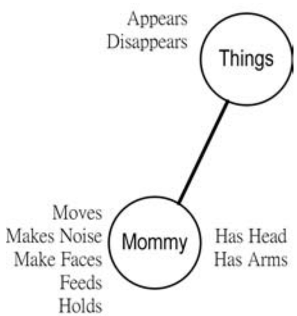
Structural Programming

- Top-down programming approach
- The top-down design decomposes a problem into modules
- Each module is a self-contained collection of steps that solves one part of the problem
- Most functions share global data
- Data moves around functions in the system
- Functions transform data in different forms
- Emphasis is on algorithms
- Structured Programming techniques
  - Rules for writing procedures for creating logically correct programs
  - For reducing logical errors
  - Help to find and correct errors

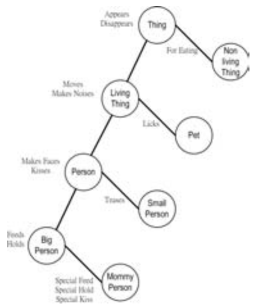
Object-Oriented Programming

- The essence of object-oriented programming is to treat data and procedures that act upon the data as a single "object"
- The "object" is a self-contained entity with its own identity and characteristics
- Emphasis is on data rather than procedures
- Objects are characterized by data structures
- Functions that operate on the data of an object are tied together in the data structures
- Objects may communicate with each other through functions
- Easy to add new objects and functions
- Bottom-up programming approach

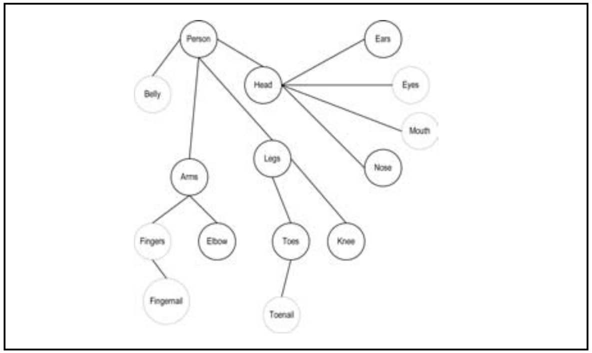
Introduction to Objects



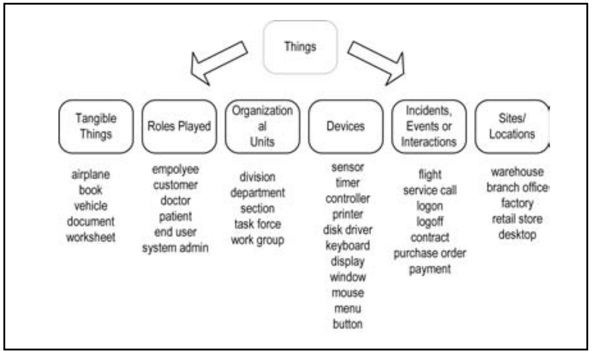
Introduction to Objects



Introduction to Objects



Introduction to Objects



Introduction to Objects

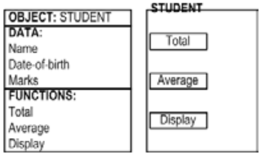
- Types of Objects in Computer Systems
  - User Interface Objects
  - Operating Environment Objects
  - Task-Related Objects
    - Document objects
    - Multimedia objects
    - Problem domain objects

Object-Oriented Programming

- Objects
  - Basic run-time entities in an object-oiented system
  - May represent a person, a place, a bank account, a table of data or any items that the program has to handle
  - May represent user-defined data such as vectors, time and lists
  - Programming problem is analyzed in terms of objects and communication between objects
  - Program objects are chosen so that they are related to real-world objects
  - Objects take up memory space

Object-Oriented Programming

- Example



- Objects interact by sending messages to one another
- “customer” and “account” are two objects
- Customer object may send a message to account object requesting for the bank balance
- Each object contain data and code to manipulate the data
- Objects can interact without knowing details of other’s data or code
- It is sufficient to know the type of message accepted and the type of response returned by the objects

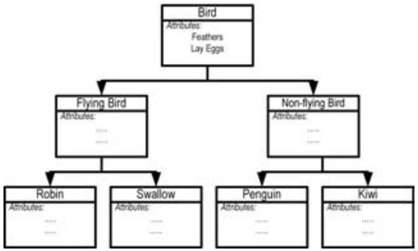
Object-Oriented Programming

- Classes
  - Objects contain data , and code to manipulate that data
  - The data and code of an object can be made a user-defied data type
  - Objects are variables of the type class
  - Once a class is defined, any number of objects can be created to the class
  - Each object is associated with the data of the type class
  - A class is a collection of objects of similar type
  - E.g. mango, apple and orange are members of the class fruit
  - Classes are user-defined data types and behave as the built-in types
  - E.g. Fruit mango;
  - The statement create an object mango belonging to the class fruit

Object-Oriented Programming

- Inheritance
  - Is the process by which objects of one class acquire the properties of objects of another class
  - Supports the concept of hierarchical classification
  - Provides the idea of reusability
    - Add additional features to an existing class without modifications
    - Deriving a new class from the existing class
    - The new class combines features of the both classes
    - Inheritance mechanism allows programmer to reuse a class

Object-Oriented Programming



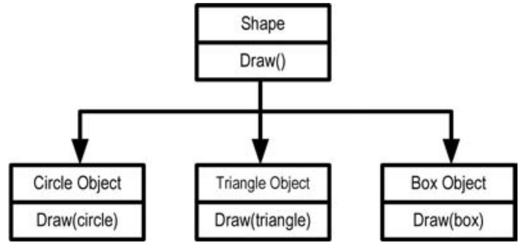
Property Inheritance

The bird “robin” is part of the class “flying bird” which is again a part of the class “bird”

Object-Oriented Programming

- Polymorphism
  - A Greek term, means the ability to take more than one form
  - An operation may exhibits different behaviors in different instances
  - The behaviors depends upon the types of data type used in the operation
  - The process of making an operator to exhibit different behaviors in different instances is known as operator overloading
  - A single function name to perform different types of tasks is known as function overloading
  - Allows objects having different internal structures to share the same external interface general class of operation may be accessed in the same manner even though specific actions associated with each operation may different

Object-Oriented Programming

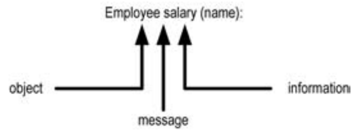


Polymorphism

Object-Oriented Programming

- Dynamic Binding
  - Binding refers to the linking of a procedure call to the code to be excuted in response to the call
  - Dynamic binding or late binding is the code associated with a given procedure call is not known until the time of the call at run-time
- Basic steps for OOP
  - Creating classes that define objects and their behavior
  - Creating objects from class definitions
  - Establishing communication among objects

Object-Oriented Programming



- Message Passing
  - A message for an object is a request for execution of a procedure
  - Invoke a function in the receiving object that generates the desired result
  - Message passing involves specifying the name of the object , the name of the function (message) and the information to be sent

Class

- A class is an extension of structure used in C
- A new way of creating and implementing user-defined data type
- C structures
  - Provide a method for packing together data of different types
  - Example:

```
Struct  student
{
        Char        name[20];
        Int         roll_number;
        Float       total_marks;
};
```

  - The structure name can be used to create variable of type student

```
Struct student A; //c declaration
```


Class

- A is a variable of type student and has three member variables can be accessed using the dot as follows:

```
Strcpy(A.name, "John");
A.roll_number = 999;
A.total_marks  = 595.5;
Final_total = A.total_marks + 5;
```

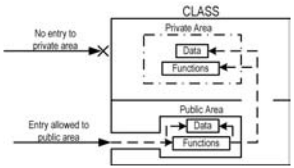
  - C++ supports all features of structures as defined in C
  - C++ has expanded its capabilities of OOP

Class

- A class is a way to bind the data and its associate functions together
- Allows the data or function to be hidden from external use
- Defined by creating a new abstract data type
- Specified by two parts:
  - Class declaration for describing the type and scope of its members
  - Class function definitions for describing how the class functions are implemented
- General form of a class declaration

```
class class_name
{     private:
           variable declarations;
           function declarations;
      public:
           variable declarations;
           function declarations;
};
```

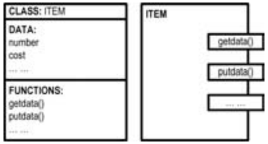

Class



- Private members can be assessed only from within the class
- Public members can be accessed from outside the class
- By default, members of a class are private
- Variables declared inside the class are known as data members
- The functions are known as member functions
- Only member functions can access to the private data members and private functions
- Public members functions and data, can be accessed from the outside class

Class Example

```
class item
{
        int number;                    // variables declaration
        float cost;                    // private by default
    public:
        void getdata(int a, float b);  // function declaration
        void putdata(void);            // using prototype
};
```



Creating Objects

- Item is a class name
- A new type identifier used to declare instances of that class type
- The identifiers contains 2 data members and 2 function members
- Data members are private by default
- Functions are public by declaration, and not defined
- We create variable of the type by using class name
  - Item x ; //memory for x is created
  - Item x, y, z; // for more than one objects
- Objects can also created when a class is defined

```
Class   item
{       ……
        ……

} x, y, z;
```


Class Members

- Accessing class members

  Format for calling member function

  object-name.function-name (actual-arguments);

  Example x.getdata (100,75.5); // 100 is the number, 75.5 is the cost

  x.putdata();

- Defining member functions
  - Outside the class definition, format:

return-type **class-name** :: function-name (argument declaration)

```
{
        Function body
}
```


Defining Member Functions

```
void item :: getdata(int a, float b)
{
        number = a;
        cost = b;
}

void item :: putdata(void)
{
        cout << "Number :" << number << "\n";
        cout << "Cost   :" << cost   << "\n";
}
```

Defining Member Functions

- Inside the class definition

```
class item
{
        int number;
        float cost;
public:
        void getdata(int a, float b);       // declaration

        // inline function

        void putdata(void);                 // definition inside the class
        {
                cout << number << "\n";
                cout << cost   << "\n";
        }
}
```


Program With Class Example (for reference)

```
#include <iostream.h>
using namespace std;
class  item
{
        int number;                     // private by default
        float cost;                     // private by default
public:
        void getdata(int a, float b);   // prototype declaration, to be defined
        // Function defined inside class
        void putdata(void)
        {
                cout << "number :" << number << "\n";
                cout << "cost   :" << cost << "\n";
        }
};
```


Program With Class Example (for reference)

```
//...... Member Function Definition .......

void item :: getdata(int a, float b)        // use membership label
{
        number = a;                         // private variables
        cost = b;                           // directly used
}
```


Program With Class Example (for reference)

```
//...... Main Program .......
int main()
{
        item x;                             // create object x
        cout << "\nobject x " << "\n";
        x.getdata(100, 299.95);             // call member function
        x.putdata();                        // call member function
        item y;                             // create another object
        cout << "\nobject y" << "\n";
        y.getdata(200, 175.50);
        y.putdata();
        return 0;
}
```

Nesting of Member Functions (for reference)

```
#include <iostream.h>

using namespace std;

class set
{
        int m, n;
public:
        void input(void);
        void display(void);
        int largest(void);
};
```


Nesting of Member Functions (for reference)

```
int set::largest(void)
{
        if(m<=n)
                return(m);
        else
                return(n);
}

void set::input(void)
{
        cout << "Input values of m and n" << "\n";
        cin >> m >> n;
}
```


Nesting of Member Functions (for reference)

```
void set:: display(void)
{
        cout << "Largest values = "
                << largest() << "\n";  // calling member function
}

int main()
{
        set A;
        A.input();
        A.display();

        return 0;
}
```


Private Member Functions (for reference)

```
class sample
{
        int m;
        void read(void);        // private member function
public:
        void update(void);
        void write(void);
};
s1.read();                      // won't work; objects cannot access
                                // private members
void sample::update(void)
{
        read();                 // simple call; no object used
}
```

Objects as Function Arguments

```
#include <iostream.h>
using namespace std;
class time
{
        int hours;
        int minutes;
public:
        void gettime(inth, int m)
        {
                hours = h;
                minutes = m;
        }
        void puttime(void)
        {
                cout << hours << " hours and ";
                cout << minutes << "minutes " << "\n";
        }
        void sum(time, time);    // declaration with objects as arguments
};
```


Objects as Function Arguments

```
void time::sum(time t1, time t2)              // t1, t2 are objects
{
        minutes = t1.minutes + t2.minutes;
        hours = minutes/60;
        minutes = minutes%60;
        hours = hours + t1.hours + t2.hours;
}

int main()
{
        time T1, T2, T3;
        T1.gettime(2,45);       // get T1
        T2.gettime(3,30);       // get T2
        T3.sum(T1,T3);                  // T3=T1+T2
        cout << "T1 = "; T1.puttime();          // display T1
        cout << "T2 = "; T2.puttime();          // display T2
        cout << "T3 = "; T3.puttime();          // display T3
        return 0;
}
```


Friendly Functions

- Non-member function cannot have an access to the private data of a class
- C++ allows the common function to be made friendly with both classes and allow the function have an access to the private data of these classes
- Format:

```
class ABC
{
        .....
        .....
public:
        .....
        .....
        friend void xyz(void);          // declaration
};
```


Friendly Functions

```
#include <iostream.h>
using namespace std;
class sample
{
        int a;
        int b;
public:
        void setvalue() {a=25; b=40; }
        friend float mean (sample s);
};
float mean(sample s)
{
        return float(s.z +s.b)/2.0;
}
```

Friendly Functions

```
int main()
{
            sample X;               // object X
            X.setvalue);
            cout << "Mean value = " << mean(X) << "\n";

            return 0;
}
```


Constructors

- A constructor is a “special” member function whose task is to initialize the objects of its class
- Its name is same as the its class name
- The constructor is invoked whenever an object is created
- A constructor is declared and defined as follows:

```
// class with a constructor

class integer
{
            int m, n;
public:
            integer(void);                          // constructor declared
            .....
            .....
};
integer::integer(void)      // constructor defined
{
            m = 0; n = 0;
}
```


Constructors

- A constructor guaranteed that an object created by the class will be initialized automatically
  integer int1:   // object int1 created
- Not only creates the object int1 of type integer but also initializes its data members m and n to zero
- A constructor that accepts no parameters is called default constructor
- for class A is A::A()
- If no such constructor is defined, the compiler supplies a default constructor
- Should be declared in the public section
- No return types and cannot return a value
- Cannot be inherited, though a derived class can call the base class constructor
- Can have default arguments
- They make “implicit calls” to the operator new and delete when memory allocation is required

Constructors

- Constructors can be passed with arguments to the constructor function when the objects are created
- Two ways:
  - By calling the constructor explicitly
    Integer int1 = integer(0,100);   // explicit call
  - By calling the constructor implicitly
    Integer int1(0,100);   // implicit call
- Constructors with default arguments
  complex(float real, float imag=0);
  complex c(2.0, 3.0)
- Default constructor   A::A()
- Default argument constructor   A::A(int=0)

- Example:

Constructors

```
#include <iostream.h>
using namespace std;
class integer
{
        int m, n;
public:
        integer(int, int);              // constructor declared
        void display(void)
        {
                cout << " m = " << m << "\n";
                cout << " n = " << n << "\n";
        }
};

integer::integer(int x, int y)          // constructor defined
{
        m = x; n= y;
}
```


Constructors

```
int main()
{
        integer int1(0,100);                    // constructor called implicitly

        integer int2 = integer(25, 75);         // constructor called implicitly

        cout << "\nOBJECT1" << "\n";
        int1.display();

        cout << "\nOBJECT2" << "\n";
        int2.display();

        return 0;
}
```


Multiple Constructors in a Class

- Overloaded Constructors
- Constructors 1, 2 and 3 can be used in the same class

```
class integer
{
        int m, n;
public:
        integer(){m=0; n=0}             // constructor 1
        integer(int a, int b)
        {m = a; n = b;}                 // constructor 2
        integer(integer & i)
        {m = i.m; n = i.n;}             // constructor 3
}
```


Multiple Constructors in a Class

Example:

```
#include <iostream.h>
using namespace std;
class complex
{
        float x, y;
public:
        complex(){ }                            // constructor no arg
        complex(float a) {x = y = a;}           // constructor-one arg
        complex(float real, float imag)         // constructor-two args
        { x = real; y = imag;}
        friend complex sum(complex, complex);
        friend void show(complex);
};
```

Multiple Constructors in a Class

```
void show(complex c)     // friend
{
        cout << c.x << " + j" << c.y << "\n";
}
complex sum(complex c1, complex c2)          // friend
{
        complex c3;
        c3.x = c1.x + c2.x;
        c3.y = c1.y + c2.y;
        return(c3);
}
```


Multiple Constructors in a Class

```
void show(complex c)     // friend
{
        cout << c.x << " + j" << c.y << "\n";
}
complex sum(complex c1, complex c2)          // friend
{
        complex c3;
        c3.x = c1.x + c2.x;
        c3.y = c1.y + C2.y;
        return(c3);
}
```


Multiple Constructors in a Class

```
int main()
{       complex A(2.7, 3.5);     // define & initialize
        complex B(1.6);          // define & initialize
        complex C;
        C = sum(A, B);           // sum() is a friend
        cout << "A = "; show(A);// show() is also friend
        cout << "B = "; show(B);
        cout << "C = "; show(C);
// Another way to give initial values (second method)
        complex P,Q,R;                  // define P, Q and R
        P = complex(2.5,3.9);    // initialize P
        Q = complex(1.6,2.5);    // initialize Q
        R = sum(P,Q);
        cout << "\n";
        cout << "P = "; show(P);
        cout << "Q = "; show(Q);
        cout << "R = "; show(R);
        return 0;
}
```


Dynamic Initialization Of Object (for reference)

```
// Long-term fixed deposit system
#include <iostream.h>

using namespace std;

class Fixed_deposit
{
        long int P_amount;              // Principal amount
        int Years;              // Period of investment
        float Rate;             // Interest rate
        float R_value;                  // Return value of amount
public:
        Fixed_deposit(){ }
        Fixed_deposit(long int p, int y, float r=0.12);
        Fixed_deposit(long int p, int y, int r);
        void display(void);
};
```

Dynamic Initialization Of Object (for reference)

```
Fixed_deposit::Fixed_deposit(long int p, int y, float r)
{
        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount;
        for(int i = 1; i <= y; i++)
                R_value = R_value * (1.0 +r);
}
Fixed_deposit::Fixed_deposit(long int p, int y, int r)
{
        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount;

        for(int i=1; i<=y; i++)
                R_value = R_value*(1.0+float(r)/100);
}
```


Dynamic Initialization Of Object (for reference)

```
void Fixed_deposit::display(void)
{
        cout << "\n"
                << "Principal Amount = " << P_ amount << "\n"
                << "Return Value    = " << R_value  << "\n";
}
```


Dynamic Initialization Of Object (for reference)

```
int main()
{       Fixed_deposit FD1, FD2, FD3;        // deposits created
        long int p;                 // principal ammount
        int y;                              // investment period, years
        float r;                            // interest rate, decimal form
        int R;                              // interest rate, percent form
        cout << "Enter amout, period, interest rate(in percent)" << "\n";
        cin >> p >> y >> R;
        FD1 = Fixed_deposit(p,y,R);
        cout << " Enter amout, perio, interest rate(decimal form): << "\n";
        cin >> p >> y >> r;
        FD2 = Fixed_deposit(p,y,r);
        cout << "Enter amout and period: << "\n";
        cin >> p >> y;
        FD3 = Fixed_deposit(p,y);                   FD2.display();
        cout << "\nDeposit 1";              cout << "\nDeposit 3";
        FD1.display();                              FD3.display();
        cout << "\nDeposit 2";              return 0;
        }
```


Destructors

- A destructor is used to destroy the created objects by its constructor
- Destructor is a member function with name same as the class name but is preceded by a tilde ~integer() { }
- Never takes any argument nor does it returns any value
- Will be invoked implicitly by the compiler upon exit from the program or function to clean up storage that no longer accessible
- Whenever **new** is used to allocate memory in the constructors, **delete** should be used to free memory

Destructors

```
#include <iostream.h>
using namespace std;
int count = 0;
class alpha
{
public:
        alpha()
        {
                count++;
                cout << "\nNo. of object created " << count;
        }

        ~alpha()
        {
                cout << "\nNo. of object destroyed " << count;
                count--;
        }
};
```

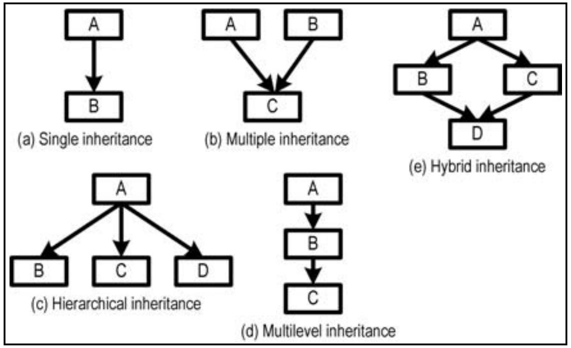

Destructors

```
int main()
{
        cout << "\n\nENTER MAIN\n";

        alpha A1, A2, A3, A4;
        {
                cout << "\n\n"ENTER BLOCK1\n';
                        alpha A5;
        }

        {
                cout << "\n\nENTER BLOCK2\n";
                alpha A6;
        }
        cout << "\n\nRE-ENTER MAIN\n";

        return 0;
}
```


Inheritance: extending class

- Reusability is an import features of OOP
- Reuse something is better than create something
- Save time and money, reduce frustration and increase reliability
- In C++, once a class is created, it can be adapted by other programmers
- The mechanism of deriving new classes from an old one is called inheritance
- The old class is called the base class and the new one is called the derived class or subclass
- Derived class inherits some or all of the traits from the base class
- A class can inherit properties from more than one class or from more than one level
- A derived class with only one base class is called single inheritance, and one with several base classes is called multiple inheritance

Inheritance: extending class



Defining Derived Classes

General Form:
Class derived-class-name : visibility-mode base-class-name
{
 ……...//
 ………// members of derived class
 ………//
};
Example:
Class ABC: private xyz // private derivation
{
 members of ABC
};
Class ABC: public xyz // public derivation
{
 members of ABC
};

Defining Derived Classes

Class ABC: XYZ //private derivation by default
{
 members of ABC
};

- Private members of a base class will never become the members of its derived class
- Inheritance can be used to modify and extend the capabilities of the existing classes

Single Inheritance (public)

```
#include <iostream.h>
using namespace std;
class B
{
        int a;                  // private not inheritable
public:
        int b;                  // public; ready for inheritance
        void get_ab();
        int  get_a(void);
        void show_a(void);
};
class D:public B        // public derivation
{
        int c;
public:
        void mul(void);
        void display(void);
};
```

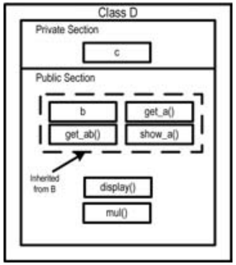

Single Inheritance (public)

```
void B::get_ab(void)
{       a=5; b=10;
}
int B::get_a()
{       return a;
}
void B::show_a()
{       cout << "a = " << a << "\n";
}
void D::mul()
{       c= b * get_a();
}
void D::display()
{       cout << "a = " << get_a() << "\n";
        cout << "b = " << b << "\n";
        cout << "c = " << c << "\n\n";
}
```

Single Inheritance (public)

```
int main()
{
        D d;

        d.get_ab();
        d.mul();
        d.show_a();
        d.display();

        d.b = 20;
        d.mul();
        d.display();

        return 0;
}
```


Single Inheritance (public)



Adding more members to a class by public derivation

Single Inheritance (private)

```
#include <iostream.h>
using namespace std;
class B
{
        int a;                  // private; not inheritable
public:
        int b;                  // public; ready for inheritance
        void get_ab();
        int  get_a(void);
        void show_a(void);
};
class D : private B     // private derivation
{
        int c;
public:
        void mul(void);
        void display(void);
};
```

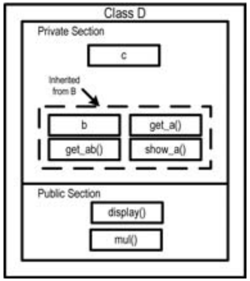

Single Inheritance (private)

```
void B::get_ab(void)
{       cout << "Enter values for a and b:";
        cin>> a >> b;
}
int B::get_a()
{       return a;
}
void B :: show_a()
{       cout << "a = " << a << "\n";
}
void D::mul()
{       get_ab();
        c = b * get_a();        // 'a' cannot be used directly
}
void D::display()
{       show_a();                       // outputs value of 'a'
        cout << "b = " << b << "\n"
                << "c = " << c << "\n\n";
}
```

Single Inheritance (private)

```
int main()
{
        D d;

        // d.get_ab(); WON'T WORK
        d.mul();
        // d.show_a(); WON'T WORK
        d.display();

        // d.b = 20    WON'T WORK; b has become private
        d.mul();
        d.diaplay();

        return 0;
}
```


Single Inheritance (private)



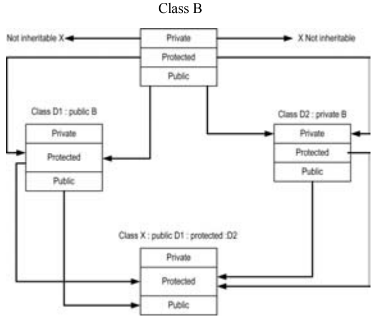
Adding more members to a class by private derivation

Making A Private Member Inheritable

- A member declared as protected is accessible by the member functions within its class and any class immediately derived from it.
- It cannot be accessed by the functions outside these two classess

```
class alpha
{
private:            //optional
        .....           // visible to member functions
        .....           // within its class
protected:
        .....           // visible to member functions
        .....           // of its own and derived class
public:
        .....           // visible to all functions
        .....           // in the program
```


Effect of Inheritance on Visibility of Members



Visibility of Inherited Members

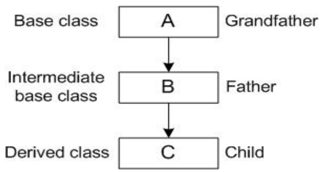
Base class visibility	Public Derivation	Private Derivation	Protected Derivation
Private --- >	Not inherited	Not inherited	Not inherited
Protected --- >	Protected	Private	Protected
Public --- >	Public	Private	Protected

Multilevel Inheritance

- A derived class with multilevel inherited is declared as follows:

```
class A {….};                    // Base class
class B: public A {…..} ;   // B derived from A
class C: public B {…..} ;   // C derived from B
```

- The process can be extended to any number of levels



Multilevel Inheritance

```
#include <iostream.h>

class student
{
protected:
        int roll_number;
public:
        void get_number(int);
        void put_number(void);
};
void student::get_number(int a)
{
        roll_number = a;
}
void student::putnumber()
{
        cout << "Roll Number: " << roll_number <, "\n";
}
```


Multilevel Inheritance

```
class test : public student                    // First level derivation
{
protected:
        float sub1;
        float sub2;
public:
        void get_marks(float, float);
        void put_marks(void);
};
void test::get_marks(float x, float y)
{
        sub1 = x;
        sub2 = y;
}
void test::putmarks()
{
        cout << "Marks in SUB1 = " << sub1 << "\n";
        cout << "Marks in SUB2 = " << sub2 << "\n";
}
```

Multilevel Inheritance

```
class result : public test            // Second level derivatikon
{
        float total;                  // private by default
public:
        void display(void);
};

void result::display(void)
{
        total = sub1 + sub2;
        put_number();
        put_marks();
        cout << "Total = " << total  << "\n";
}
```


Multilevel Inheritance

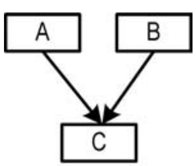
```
int main()
{
        result student1;              // student1 created
        student1.get_number(111);
        student.get_marks(75.0, 59.5);

        student1.display();

        return 0;
}
```


Multiple Inheritance

A derived class with multiple base classes is declared as follows:

```
Class D: visibility A, visibility B
{
        ………….
        ………….        (Body of D)
        ………….
};
```

Where visibility may be either public or private



Multiple inheritance

Multiple Inheritance

```
#include <iostream.h>
using namespace std;
class M
{
protected:    int m;
public:       void get_m(int);
};
class N
{
protected:    int n;
public:       void get_n(int);
};
class P : public M, public N
{
public:       void display(void)
};
```

Multiple Inheritance

```
void M::get_m(int x)
{
        m = x;
}

void N::get_n(int y)
{
        n = y;
}

void P::display(void)
{
        cout << "m = " << m << "\n";
        cout << "n = " << n << "\n";
        cout << "m*n = " << m*n << "\n";
}
```

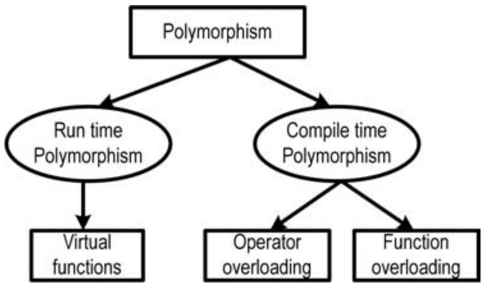

Multiple Inheritance

```
int main()
{
        P p;

        p.get_m(10);
        p.get_n(20);
        p.display();

        return 0;
}
```


Polymorphism

- Means “one name, multiple forms”
- Implemented using overloaded functions or operators
- Information of the overloaded member functions is known to the compiler at the compiler time for selecting the appropriate call function. This is called early binding or static binding or static linking, also known as compile time polymorphism
- If appropriate member function is selected while the program is running. It is known as run time polymorphism
- At run time, when it is considering the class objects, the appropriate function is invoked. The function is linked with a particular class much later after compilation, the process is called late binding, or dynamic binding.

Polymorphism



Polymorphism

```
#include <iostream.h>                    // Pointers to Objects
using namespace std;
class item
{
        int code;
        float price;
public:
        void getdata(int a, float b)
        {
                code = a;
                price = b;
        }
        void show(void)
        {
                cout << "Code : " << code << "\n";
                cout << "Price: " << price << "\n";
        }
};
const int size = 2;
```


Polymorphism

```
int main()                               //    Pointers to Objects
{
        item *p = new item [size];
        item *d = p;
        int x, i;
        float y;
        for(i=0; i<size; i++)
        {       cout << "Input code and price for item" << i+1;
                cin >> x >> y;
                p->getdata(x,y);
                p++;
        }
        for(i+0; i<size; i++)
        {       cout << "Item: " << i+1 << "\n";
                d->show();
                d++;
        }
        return 0;
}
```

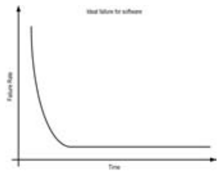

Software Engineering

- Basic Concepts
- Software Process Models
- Waterfall Model
- Fountain Model
- Object-Oriented Analysis
- Prototyping Models

Hardware Failure

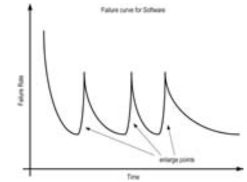
- Here is the bathtub curve.
- It represents hardware failure rate as a function of time
- Design and manufacturing faults cause parts to fail early, defects are corrected and failure rate stays low for some period then rises as dust, vibration, wear and abuse accumulate

Software Failure



- Software does not wear out, so its curve should look like this
- Unfortunately, software maintenance involves change
- Each change will increase the likelihood of failures

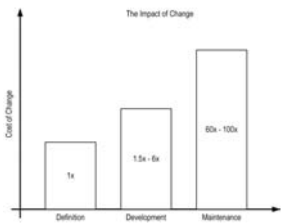
Software Failure



- Changes introduce new flaws
- The failure rates of the new and existing components are cumulative
- We need to design and build it correctly the first time to reduce the need for changes later
- We need to build it in a maintainable way

Cost of Poor Design

- Reliability is not the only reason to design software correctly
- The cost of change increases dramatically through the project



Common Myth

- A general statement of objectives is sufficient to begin writing programs, we can fill in the details later
- The reality
  - Poor up front definition is a major cause of failed software development
  - Formal and detailed description of the domain, function, performance, interfaces and validation criteria is essential
  - You can only create these after thorough communication between customer and developer

Software Engineering

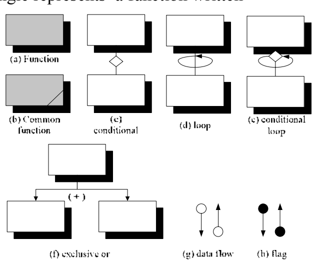
- The difference between software engineering and programming is that software engineering is concerned with the complete lifecycle of the software, from initial proposal to retirement
- Programming is merely a task that is performed during some parts of the process
- Software engineering is the establishment and use of sound engineering principles in order to obtain economical software that is reliable and works efficiently on real machines
- Software engineering encompasses a set of three key elements: methods, tools and procedures

Structured Charts

- Primary design tool for a program
- An analogy that helps you understand the importance of designing before coding
- Used also as a program review process called a structured walk-through
  - Ensures that you understand how your program fits into the system by communicating the design to the team
  - Validates the design
  - Ensures that the final program will be robust and as error-free as possible

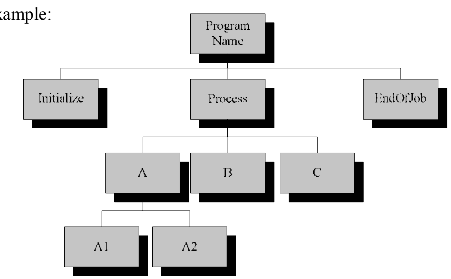
Structured Charts: Rules and Symbols

- Each rectangle represents a function written



Reading Structured Charts

- Example:



Reading Structured Charts

- Top-down, from left to right
- Three subfunctions: Initialize, Process, and EndOfJob
- First call: Initialize
- After Initialize is complete, calls Process
- After Process is complete, calls EndOfJob

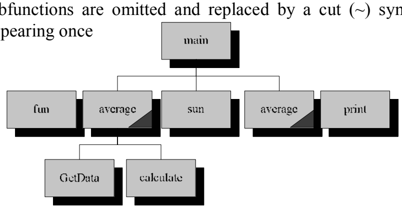
Structured Charts show only function flow; they contain no codes.

Reading Structured Charts

- Often a program will contain several calls to a common function
- A cross-hatch or shading in the lower right corner of a rectangle identifies a common function
- If the common function is complex and contains subfunctions, these subfunctions need to be shown only once

Reading Structured Charts

- Example:
  - average is a common function in two different places of the program
  - The subfunctions are omitted and replaced by a cut (~) symbol after appearing once



Summary of rules

1. Each rectangle in a structured chart represents a function written by the programmer. Standard C++ functions are not included.
2. The name in the rectangle is an intelligent name that communicates the purpose of the function. It is the name that will be used in the coding of the function.
3. The function chart contains only functions flow. No code is indicated.

Summary of rules

4. Common functions are indicated by a cross-hatch or shading in the lower right corner of the function rectangle.
5. Common calls are shown in a structured wherever they will be found in the program. If they contain subfunction calls, the complete structure need to be shown only once.
6. Data flows and flags are optional. When used, they should be named.
7. Input flows and flags are shown on the left of the vertical line; output flows and flags are shown on the right.

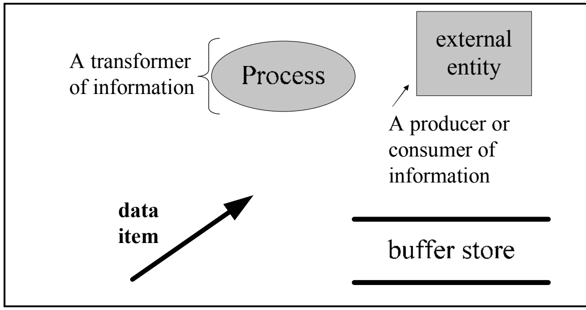
Data Flow Diagram (DFD)

- Most basic diagrams in software development
- Shows the flow of the data among a set of components
- Components:
  - Tasks
  - software components
  - abstractions of functionality

Rules of DFD

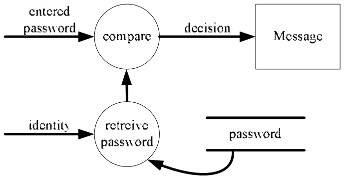
1. Boxes and processes must be verb phrases
2. Arcs represent data and must be noun phrases
3. Control is not shown. Some sequencing may be inferred from the ordering
4. A process may be a one-time activity, or it may imply a continuous processing
5. Two arcs coming out a box may indicate that both outputs or only either one output of the two are produced

Notations used in DFD



Notations used in DFD

- Example: The software *retrieves* the user's password based on the **identity** claimed by an unknown user and *compares* it against an **entered password** from the unknown user. A **message** stating the decision will be shown.



Entity Relationship Diagram

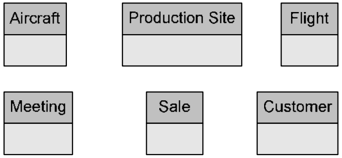
- A graphical representation of an organization's data storage requirements.
- Abstractions of the real world
- Simplifies the problem to be solved

Entity Relationship Diagram

- Usage:
  - Identify the data that must be captured, stored and retrieved in order to support the business activities performed by an organization
  - Identify the data required to derive and report on the performance measures that an organization should be monitoring
- Three different components:
  - Entities
  - Attributes
  - Relationships

Entities

- Anything which an organization needs to store data about
- Represented on the diagram by labelled boxes

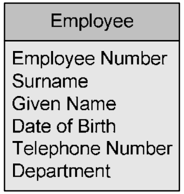


Entities

- Represents collections of things
- Example:
  - an EMPLOYEE entity might represent a collection of all employees that work for an organization.
  - Individual members (employees) of the collection are called occurrences of the EMPLOYEE entity.
- The available space for naming the entity is restricted to the size of the box
- Entities should always have detailed descriptions
  - Short paragraphs of text describing the entity
  - A lengthy description may be required for some important entities

Attributes

- Entities are further described by their attributes
- Sometimes also known as data elements
- Smallest units of data can be described in a meaningful manner
- Example:

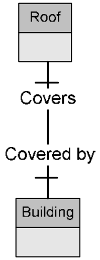


Relationships

- A meaningful relationship exists between two different types of entity
  - EMPLOYEEs work in a DEPARTMENT
  - LAWYERs advise CLIENTS
  - EQUIPMENT is allocated to PROJECTs
  - TRUCK is a type of VEHICLE
- Three types of relationship:
  - One-to-One Relationships
  - One-to-Many Relationships
  - Many-to-Many Relationships

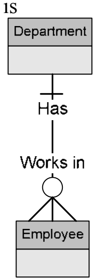
One-to-One Relationship

- A single occurrence of an entity related to just one occurrence of a second entity
- Example:
  - A ROOF covers one BUILDING; a BUILDING is covered by one ROOF
  - Shown on the diagram by a line connecting the two Entities



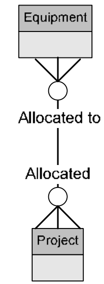
One-to-Many Relationships

- Takes place when a single occurrence of an entity is related to many occurrences of a second entity
- Example:
  - An EMPLOYEE works in one DEPARTMENT; a DEPARTMENT has many EMPLOYEEs
  - Shown on the diagram by a line connecting the two entities with a crows feet symbol denoting the “many” end of the relationship



Many-to-Many Relationships

- Takes place when many occurrences of an entity are related to many occurrences of a second entity
- Example:
  - EQUIPMENT is allocated to many PROJECTs; a PROJECT is allocated many times of EQUIPMENT
  - Shown on the diagram by a line connecting the two entities

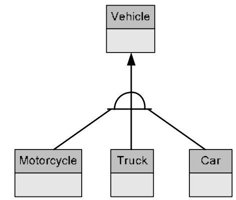


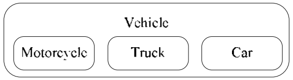
Entity Sub-Types

- Useful to generalize about a group of Entities which have similar characteristics
- Example:
  - a VEHICLE is a generalization of a CAR, a TRUCK and a MOTORCYCLE.
  - a CAR is a specialized type of VEHICLE
  - a TRUCK is a specialized type of VEHICLE
  - a MOTORCYCLE is a specialized type of VEHICLE

Entity Sub-Types

- Two common styles to show Entity Sub-Types





Subject Areas

- Subdivide a large, complex entity relationship diagram into a number of Subject Areas
- Each Subject Area focuses on a single aspect of the problem
- Example:
  - A model depicting the Human Recourse data required by an organization should be subdivided in to the following Subject Areas
  - Recruitment
  - Safety
  - Payroll
  - Restoring

Software Development

- Software Engineering Methods
  - Software engineering methods provide the technical "how to's" for building software
  - Methods encompass a broad array of tasks that include:
    - Project planning & estimation
    - System and software requirement analysis
    - Data structure design
    - Program architecture
    - Algorithm procedure
    - Coding
    - Testing
    - maintenance

Software Development

- Software Engineering Tools
  - Software engineering tools provide automated or semi-automated support for methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering (CASE), is established
- Software Engineering Procedures
  - Procedures define the sequence in which methods will be applied, the deliverables (documents, reports, forms and etc) are required
  - The controls that help assure the quality and coordinate change, and the milestones that enable software managers to assess progress

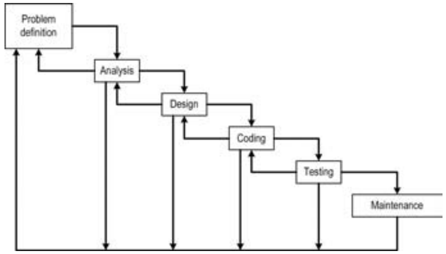
Software Development

- There exist a number of Software Engineering Paradigms
- The selection of a paradigm depends on the nature of the application, the programming language used, and the controls and deliverables required
- The development of a successful systems depends on the use of appropriate methods, techniques and the developer's commitment to the objectives of the system

Software Development

- A successful system must:
  - Satisfy the user requirements
  - Be easy to understand by the users and operators
  - Be easy to operate
  - Have a good user interface
  - Be easy to modify
  - Be expandable
  - Have adequate security controls against misuse of data
  - Handle the errors and exceptions satisfactorily
  - Be delivered on schedule within the budget

Procedure-Oriented Paradigm



Waterfall Model (classic software development life cycle)

Procedure-Oriented Paradigm

- Problem Definition
  - Requires a precise definition of the problem in user term
  - A clear statement of the problem is important to the success of the software
  - Helps developer and user to have better understanding of the problem
- Analysis
  - Detailed study of users' requirement and software
    - Inputs, processes, outputs, constraints
- Design
  - System design concepts: data structure, software architecture and algorithm
  - Translates the requirements into software representation

Procedure-Oriented Paradigm

- Coding
  - Translate design into machine code
  - More detailed design, the easier to code and the better its reliability
- Testing
  - Twst for correctness of the code and rsults
  - Involve each individual unit and the system
  - Require a detailed plan to what, when and how to test
- Maintenance
  - Ensures changes due to user's requirement, operating environment or software bugs

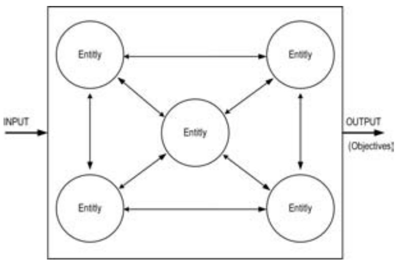
Water-fall Model life Cycle Output

Phase	Output
Problem definition (why)	• Problem statement sheet • Project request
Analysis (what)	• Requirement document • Feasibility report • Specifications document • Acceptance test criteria
Design (how)	• Design documentation • Test class design
Coding (how)	• program document • test plan • user manual
Testing (what and how)	• tested code • test results • system manual
Maintenance	• maintenance log sheets • version documents

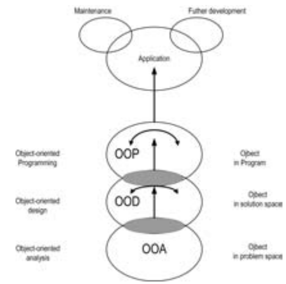
Object-Oriented Paradigm

- Draws heavily on the general systems theory as a conceptual background
- A system can be viewed as a collection of entities that interact together to accomplish certain objectives
- Entities represent physical objects such as equipment and people, and abstract concepts such as data files and functions
- Emphasis on the objects that encapsulate data and procedures
- Objects play the central role in all stages of software development
- Exists a high degree of overlap and iteraction between stages
- “Fountain Model” in place of the classic “Water-fall” model”

Object-Oriented Paradigms



Fountain Model

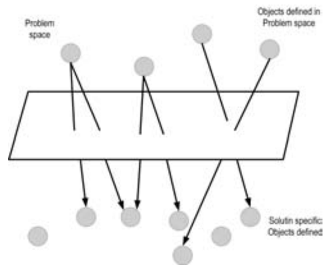


Inter-relationship of entities

Fountain Model

- Development reaches a higher level only to fall back to a previous level and then again climbing up
- Object-Oriented analysis (OOA) refers to the methods of specifying requirements of the software in terms of real-world objects, their behavior and their interactions
- Object-Oriented design (OOD) turns software requirement into specifications for objects and derives class hierarchies from the created objects
- Object Oriented programming (OOP) refers to the implementation of the program using objects
- A clear and well-organized statement of the problem is built into the application after developing specifications of the objects in the problem space
- The objects form a high-level layer of definitions
- Other objects are identified during the refinement of the application objects
- All phases work closly. One phase, problem domain objects are identified while additional objects are specified in next phase. The design process is repeated

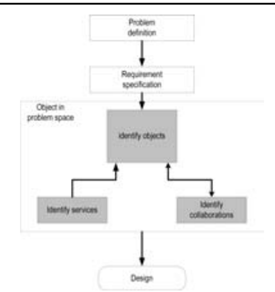
Object Specification Layers



Object-Oriented Analysis

- Provides us with a simple, powerful, mechanism for identifying objects, the building block of the software to be developed
- Is concerned with the decomposition of a problem into its component parts and establishing a logical model for the system functions
- Steps:
  - Understand the problem
  - Write user requirement and software
  - Identifying objects and their attributes
  - Identifying the object services
  - Establish inter-connections between the objects for services required and rendered.

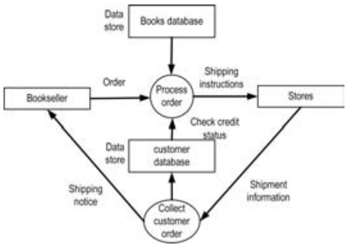
Object-Oriented Analysis



Object-Oriented Analysis

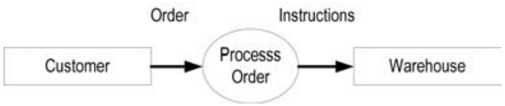
- Objects Identification for real-world objects and abstract objects
- Looking for objects by analyzing the applications
- Data Flow Diagram
  - Indicating data flow from one point to another in the system
  - The boxes and data stores are objects
  - The process bubbles corresponding to the procedures

Data Flow Diagram



Order processing and shipping for a publishing company

Data Flow Diagram



Fundamental Data Flow Diagram

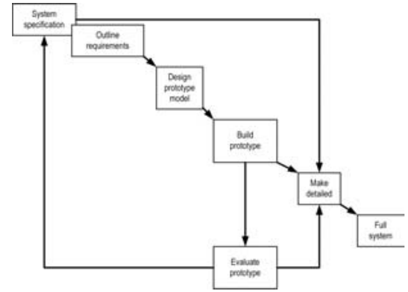
Prototyping Paradigm

- Real-world application problems are complex, structure is too large to work out the precise requirement
- Build and test a working model of the system before working on the complete system
- The model is known as prototype, and the process is prototyping
- Object-Oriented analysis and design approach is evolutionary and is suitable for prototyping paradigm
- A prototype is a scaled down version of the system and may not have stringent performance and criteria and resource requirement
- Developer and customer need to agree a “outline specifications” of the system for the prototype

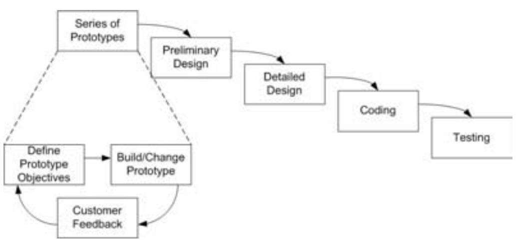
Prototyping Paradigm

- The prototype is built and evaluated for its performance
- Prototype provide the experimental evaluation of the system structure, internal design, hardware requirements and the final system requirements
- Benefits of using prototype approach:
  - Produce understandable specifications which is almost correct and complete
  - User can understand what is offering
  - Maintenance changes can be minimized
  - Developer can work on a set of tested and approved specifications

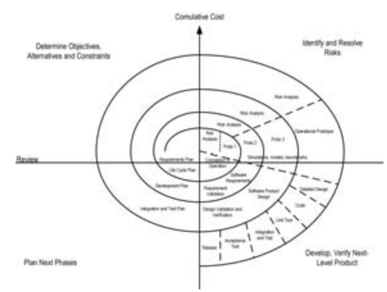
Prototyping Paradigm



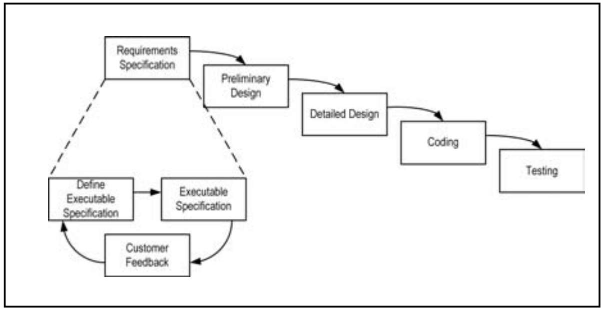
Rapid Prototyping



Spiral Life Cycle Model



Operational Specification



Reference

1. B.W. Kenighan, D.M. Ritchie, "The C Programming Language", 2nd Edition, Prentice Hall, 1998.
2. Clovis L. Tondo, Scotte Gimpel, " The C Answer Book, Solutions to the Exercises in the C Programming Language, Second Edition by Brian W. Kernighan", @nd Edition, Prentice Hall, 2000.
3. Behrouz A, Forouzan, " Computer Science: a structured programming approach using C", 2nd Edition, Brooks/Cole, 2001.
4. Byron Gottfried, "Schaum's Outlines Series - Programming with C", 2nd Edition, McGraw Hill, 1996.
5. John R. Hubbard, "Schaum's Outlines Series - Programming with C++", 2nd Edition, McGraw Hill, 2000.
6. David Gustafson, "Schaum's Outlines Series - Software Engineering", 1st Edition, McGraw Hill, 2002.
7. R. Johnsonbaugh, Martin Kalin, "Object-Oriented Programming in C++", 2nd Edition, Prentice Hall, 2000.
8. John W. Satzinger, "The Object-Oriented Approach: Concepts, System Development, and Modeling with UML", 1st Edition, Prentice Hall, 2001.
9. E. Balagurusamy, "Object-Oriented Programming with C++", 2nd Edition, MCGraw Hill, 2001.
10. David Gustafson, "Theory and problems of Software Engineering", McGraw-Hill, 2002.
11. Roger S. Pressman, "Software Engineering: A practitioner's Approach", 5th edition, McGraw-Hill, 2001.
12. http://msdn.microsoft.com
13. http://www.compilers.net/Dir/Free/Compilers/CCpp.htm
14. http://www.mit.edu/iap/cccc/c-refcard-letter.pdf
15. http://www.acm.uiuc.edu/webmonkeys/book/c_guide/
16. http://www.cs.colorado.edu/~eliuser/c_html/c.html#s1