Object-Oriented Programming with C++

Mostafa M. Aref

University of Bridgeport
Department of Computer Science & Engineering
Problem-Solving

- Problem - Reasoning - Solution - Test
- Analytic Approach
- Algorithmic Approach
  - Input, Process, Output
  - Algorithm is a sequence of executable instructions
    - no ambiguity (instruction or sequence)
    - finite (steps or execution)
  - Sequence: I/O, variables assignment
  - Selection: If . . . Else
  - Repetition: looping
  - Condition Looping: while or until
- Software Engineering
  - Requirement Specification
  - Analysis (Input, Output, Formula, Units)
  - Design (algorithm, verification)
  - Implementation (Language)
  - Testing
- Waterfall Model
Evolution of Object-Oriented Programming

- **Variables**
  - A variable is a value that can change, depending on conditions or on information passed to the program.

- **Data types**
  - A set of values from which a variable, constant, function, or other expression may take its value. A type is a classification of data that tells the compiler or interpreter how the programmer intends to use it.

- **User defined data types**

- **Abstract Data types**
  - A type whose internal form is hidden behind a set of access functions. Objects of the type are created and inspected only by calls to the access functions. This allows the implementation of the type to be changed without requiring any changes outside the module in which it is defined.
  - Abstraction
  - Encapsulation

- **Object-Oriented Programming**
Object-Oriented Programming

Advantages of Object-Oriented Programming
- Simplicity
- Modularity
- Modifiability
- Extensibility
- Flexibility
- Maintainability
- Reusability

Object-Oriented features

Abstraction: The process of capturing the essential features and ignoring the detail

Encapsulation: information hiding mechanism
- Object: An entity which has both variables and methods
  - Variables comprise the state of the object
  - Methods are mechanisms for accessing or changing the state of the object
  - Public variables or methods
  - Private variables and methods
- Class: a template that can be used to create many objects with a different name and identity yet sharing the method code and shared variables

Instantiation: The process of creating objects using the description of a class
  MyClass m1 = new MyClass();
- Message Passing: an object sends a message to another requesting that a service be performed.

Inheritance

Polymorphism: the same message is sent to a collection of objects & each object will be able to respond in its own way
## Object-Oriented Design

### Object-Oriented Strategies

<table>
<thead>
<tr>
<th>Design Strategy</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction</td>
<td>Simplifying to its essentials the description of a real-world entity</td>
</tr>
<tr>
<td>Separation</td>
<td>Treating independently &quot;what&quot; an entity does from &quot;how&quot; it does it</td>
</tr>
<tr>
<td>Composition</td>
<td>Building complex &quot;whole&quot; systems by assembling simpler &quot;parts&quot; in one of two basic ways: Association, Aggregation</td>
</tr>
<tr>
<td>Generalization</td>
<td>Identifying common elements among different entities in one of three ways: hierarchy, polymorphism, patterns</td>
</tr>
</tbody>
</table>

**Design Strategies**

- Abstraction
- Separation
- Generalization

**Mapping Abstraction and Separation to Classes and Objects**

- Composition
  - Association
  - Aggregation

- Generalization
  - Hierarchy
  - Polymorphism
  - Patterns

**Using Existing Classes**

- Single
- Several

**Creating New Classes**

- Single
- Related

**Patterns of Classes**
Abstraction

Object-Oriented Strategies

- Well named
- Coherent
- Accurate
- Minimal
- Complete
### Separation

<table>
<thead>
<tr>
<th>What</th>
<th>Goals (Interface, specification, requirement)</th>
<th>Policy</th>
<th>Product</th>
<th>Ends</th>
</tr>
</thead>
<tbody>
<tr>
<td>How</td>
<td>Plans</td>
<td>Mechanism</td>
<td>Process, Implementation</td>
<td>Means</td>
</tr>
</tbody>
</table>

- **Software Interface**: the external, visible aspects of a software artifact by which the behavior of the artifact is elicited.
- **Software Implementation**: The programmed mechanism that realizes the behavior implied by an interface.
- **Separation**: In software systems, the independent specification of a software interface and one or more software implementations of that interface.

![Separation Diagram](image_url)

**Diagram Explanation**:
- **visible**: Interface
- **hidden**: Implementation

The diagram illustrates the separation of concerns between the interface and its implementations, showing how different interfaces can be realized by various implementations.
Classes, Objects, and Abstract Interfaces

real-world

abstraction

software

entity → attributes → \{data, data,...\}

behavior → \{method, method,...\}

interface

object

implementation

SalesPerson

private

name

commissionRate

totalSales

public

sellCar

reportSales

John Smith
16%
$250,000

Jane Brown
16%
$275,000

sellCar

reportSales

class

instance of

abstract interface

object

implements

class

instance of
Concept of Composition

- **Composition**: An organized collection of components interacting to achieve a coherent, common behavior.

- **Association**: A composition of independently constructed and externally visible parts.
- **Advantages of an association**:
  - Parts may be shared between different compositions.
  - Parts in an association can be dynamically changed.

- **Aggregation**: A composition that encapsulates (hides) the parts of the composition.

<table>
<thead>
<tr>
<th>Object</th>
<th>Sub-Object</th>
<th>Sub-Sub-Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automobile</td>
<td>Engine</td>
<td>Pistons</td>
</tr>
<tr>
<td>Computer</td>
<td>Mother board</td>
<td>Processor chip</td>
</tr>
<tr>
<td>Molecules</td>
<td>Atoms</td>
<td>Quarks</td>
</tr>
</tbody>
</table>
Generalization

- **Generalization**: The identification, and possible organization, of common properties of abstractions. Object-Oriented Strategies
- **Hierarchy**: a generalization in which abstractions are organized into a directed acyclic graph whose arcs denote an "is-a" relation between a more generalized abstraction and the one or more derived specializations.

**Four major purposes**
- It provides a form of knowledge representation.
- The names of the intermediate levels in the hierarchy provide a vocabulary that can be used among developers and between developers and domain experts.
- The hierarchy can be extended by adding new specializations at any level.
- new attributes and behavior can be added easily to the proper subset of specializations.
Generalization

- **Polymorphism**: the ability to manipulate objects of distinct classes using only knowledge of their common properties without regard for their exact class.

![Diagram of Polymorphism](image)

Dynamic Binding

- **Design pattern**: A named generalization describing the elements and relationships of a solution for a commonly occurring design problem.

![Diagram of Client-Server Pattern](image)

Client-Server Pattern
The structures of objects and classes are related to three of the four design strategies and one of the most important software engineering goals, reusability.

Interfaces and inheritance relate to all of the software engineering goals, indicating the importance of understanding how to exploit these structures in creating quality software systems.
Unified Modeling Language (UML)

- A model is a simplification of reality.
- We build models so that we can better understand the system we are developing
  - Models help us to visualize a system as it is or as we want it to be.
  - Models permit us to specify the structure or behavior of a system.
  - Models give us a template that guides us in constructing a system.
  - Models document the decisions we have made.
- We build models of complex systems because we cannot comprehend such a system in its entirety
- Principles of Modeling
  - The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped
  - Every model may be expressed at different levels of precision
  - The best models are connected to reality
  - No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models
- The UML is a language for: Visualizing, Specifying, Constructing and Documenting (Requirements, Architecture, Design, Source code, Project plans, Tests, Prototypes, Releases)
Building Blocks of the UML

- **Things**
  - Structural things: seven elements--classes, interfaces, collaborations, use cases, active classes, components, and nodes
  - Behavioral things: two elements--interactions and state machines
  - Grouping things: Packages
  - Annotational things: a note.

- **Relationships:** There are four kinds: Dependency, Association, Generalization, Realization.

- **Diagrams:** there are nine: Class diagram, Object diagram, Use case diagram, Sequence diagram, Collaboration diagram, Statechart diagram, Activity diagram, Component diagram, and Deployment diagram

- **Rules of the UML:** The UML has semantic rules for
  - Names - What you can call things, relationships, and diagrams
  - Scope - The context that gives specific meaning to a name
  - Visibility - How those names can be seen and used by others
  - Integrity - How things properly and consistently relate to one another
  - Execution - What it means to run or simulate a dynamic model
Basic Structural Modeling

- **Classes**
  - Modeling the vocabulary of a system
  - Modeling the distribution of responsibilities in a system
  - Modeling nonsoftware things
  - Modeling primitive types

- **Relationships**
  - Dependency, generalization, and association relationships
    - A dependency is a using relationship that states that a change in specification of one thing may affect another thing that uses it but not necessarily the reverse.
    - A generalization is a relationship between a general thing and a more specific kind of that thing.
Building Blocks of the UML

- An association is a structural relationship that specifies that objects of one thing are connected to objects of another.

- When a class participates in an association, it has a specific role that it plays in that relationship (Multiplicity, Aggregation)

Common Modeling Techniques
- Modeling simple dependencies
- Modeling single inheritance
- Modeling structural relationships
- Creating webs of relationships

A diagram is a graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).
A well-structured class diagram
- Is focused on communicating one aspect of a system's static design view.
- Contains only elements that are essential to understanding that aspect.
- Provides detail consistent with its level of abstraction, with only those adornments that are essential to understanding.
- Is not so minimalist that it misinforms the reader about important semantics.
Class definition and Objects

- **Data Abstraction and Encapsulation**
  - **Classes and Objects**

```cpp
class Account // class name
{
  public:
    Account(); // constructor
    Account(unsigned n, double b); // constructor
    void deposit(double amt); // deposit amt into this account
    bool withdraw(double amt); // withdraw amt from this account
    double balance(); // balance inquiry
    unsigned id(); // account number inquiry
    bool transfer(double amt, Account& acct);

  private:
    unsigned acct_no; // account number
    double acct_bal; // current balance
};
```

- **Creating Objects**

```cpp
Account susan(5551234, 600.0);
Account Jack;
Account::Account(unsigned id, double amt)
{
  acct_no = id;
  acct_bal = amt;
}
```

- **Information Hiding**
  - **Public**
  - **Private**

```cpp
susan.deposit(25.60);
susan.bal = susan.balance();
susan.withdraw(25.0);
from.acct_bal -= 30.50; // private
to.acct_bal += 30.50; // private
```

- **Member Access Notation**

```cpp
acnt_ptr = &susan;
acnt_ptr -> deposit(11.79);
```
Efficiency Issues of Objects

- Passing and returning Objects by reference
- Object reference as const Parameter
- const methods
- Overloading methods

● A time Stamp Class

class TimeStamp {
public:
    void set( long s = 0 ) { if ( s <= 0 ) stamp = time( 0 );
        else stamp = s;  }
    time_t get() const { return stamp; }
    string getAsString() const { return extract( 0, 24 ); }
    string getYear() const { return extract( 20, 4 ); }
    string getMonth() const { return extract( 4, 3 ); }
    string getDay() const { return extract( 0, 3 ); }
    string getHour() const { return extract( 11, 2 ); }
    string getMinute() const { return extract( 14, 2 ); }
    string getSecond() const { return extract( 17, 2 ); }
private:
    string extract( int offset, int count ) const {
        string timeString = ctime( &stamp );
        return timeString.substr( offset, count );
    }

    time_t stamp;  }

● Constructors and the Destructor
- Array of classes and the default constructor

    unsigned count = 0;
    class C {
    public:
        C() { cout << "Creating C" << ++count << \n'; }  }
    C ar[ 1000 ];
    produces the output
    Creating C1
    Creating C2
    ...
    Creating C999
    Creating C1000
A Task Class

class Task {
public:
    Task( const string& ID ) {
        setId( ID );
        logFile = "log.dat";
        setST();
        ft = st; // no duration yet
    }
    Task( const char* ID ) {
        setId( ID );
        logFile = "log.dat";
        setST();
        ft = st; // no duration yet
    }
    ~Task() {
        logToFile();
    }
    void setST( time_t ST = 0 ) {
        st.set( ST );
    }
    time_t getST() const {
        return st.get();
    }
    string getStrST() const {
        return st.getAsString();
    }
    void setFT( time_t FT = 0 ) {
        ft.set( FT );
    }
    time_t getFT() const {
        return ft.get();
    }
    string getStrFT() const {
        return ft.getAsString();
    }
    void setId( const string& ID ) {
        id = ID;
    }
    void setId( const char* ID ) {
        id = ID;
    }
    string getID() const {
        return id;
    }
    double getDU() const {
        return difftime( getFT(), getST() );
    }
    void logToFile() {
        ofstream outfile( logFile.c_str(), ios::app );
        outfile << "ID: " << id << '
';
        outfile << "ST: " << getStrST();
        outfile << "FT: " << getStrFT();
        outfile << "DU: " << getDU();
        outfile.close(); //*** just to be safe!};
private:
    Task(); // default constructor explicitly hidden
    TimeStamp st; TimeStamp ft; string id; string logFile;
};

int main() {
    time_t now = time( 0 );
    Task t1( "Defrost pizzas" ), t2( "Open drink" ),
            t3( "Eat pizzas and drink" );
    t1.setST( now );
    t1.setFT( now + 3600 ); // an hour from now
    t2.setST( t1.getFT() ); // when pizzas defrosted
    t2.setFT( t2.getST() + 2 ); // fast work
    t3.setST( t2.getFT() + 1 ); // slight delay
    .setFT( t3.getST() + 7200 ); // leisure meal
    return 0; };

Constructors and the Destructor

- Restricting Object Creation Through Constructors
  - If a class explicitly declares *any* constructor, the compiler does *not* provide a public default constructor. In this case, the programmer must provide a public default constructor if desired.
  - If a class declares a nonpublic default constructor, the compiler does *not* provide a public default constructor.

- The Copy Constructor

```cpp
Person q("Mickey"); // constructor.
Person p = q;       // copy constructor.
Person r(p);        // copy constructor.
Person(const Person & p); // copy constructor
```

- Don't write copy constructor if shallow copies are ok

```cpp
class Vector3 {
  double  v[3];
public:
  Vector3()  { v[0] = v[1] = v[2] = 0.0; } // default constructor
  Vector3(double val)    { v[0] = v[1] = v[2] = val;    }
  Vector3(double x, double y, double z)
  { v[0] = x; v[1] = y; v[2] = z;    }
  Vector3(const Vector3 &x) // copy constructor
  { for (int i = 0; i < 3; i++)  v[i] = x.v[i];    }  
};
Vector3 *vp1 = new Vector3;
Vector3 *vp3 = new Vector3(1, 2, 3);
Vector3 *vp4 = new Vector3(*vp3);
```

- Disabling Passing and Returning by Value for Class Objects

- Convert Constructors
- Constructor Initializers
- Constructors and the Operators new and new[]
- The Destructor
- Static data and function members
  - A static *data member* may be *declared* inside the class declaration, as shows. However, such a static data member still must be *defined*. 

# Function Calls and Argument Passing

**Member Functions**

```cpp
#include <iostream>
#include "Account.h"

bool Account::transfer(double amt, Account& target)
{
    if( amt > target.acct_bal )
        return false; // failure
    target.acct_bal -= amt; // charge target account (a)
    acct_bal += amt; // credit host account
    return true; // success
}
```

**Reference Parameters**

**Inline Functions**

```cpp
inline int MAX(int a, int b) {return(a>b?a:b);}
```

**Command-Line Arguments**

```cpp
int main(int argc, char *argv[])
```

**Environment Variables**

**The string CLASS**

**Problem Solving With Objects**

```cpp
class Vector2D
{
    public:
        Vector2D( ) {}
        Vector2D(float a, float b) { x=a; y=b; }
        Vector2D difference(Vector2D& v);
        float inner(Vector2D& a);
        bool isParallel(Vector2D& v);
        bool isPerpendicular(Vector2D& v);
        bool nonzero( ) { return ( x != 0.0 || y != 0.0 ); }
        void display( ); /* other members not shown */

    private:
        float x, y;
};
```
```
#include <iostream>
#include "Vector2D.h"

float Vector2D::inner(Vector2D& v)
{  return(x * v.x + y * v.y); }

void Vector2D::display()
{   std::cout << "(" << x << " , "  << y << ")";;

Vector2D Vector2D::difference(Vector2D& v)
{   Vector2D tmp; tmp.x = x - v.x;
    tmp.y = y - v.y;  return tmp;}

inline float ABS(float x) {  return (x > 0 ? x : -x); }

bool Vector2D::isPerpendicular(Vector2D& v)
{  return ( nonzero() && v.nonzero()
    && ABS(inner(v)) < 0.0000001 ); }

Vector2D getVec(int i)  // input a point as vector
{  float x,y;
    std::cout << "x" << i << " = ";
    std::cin >> x;
    std::cout << "y" << i << " = ";
    std::cin >> y;
    return Vector2D(x,y); }

int main()
{  std::cout << "Enter vertices 0,1,2,3 "  << std::endl;
    Vector2D p[4];  // vector array
    for ( int i = 0; i < 4; i++) p[i] = getVec(i);
    Vector2D u = p[0].difference(p[3]);  // vector difference
    Vector2D v;
    for (int i = 0; i < 3; i++)  // process all sides
    {  v = p[i+1].difference(p[i]);  // vector difference
        if ( ! u.isPerpendicular(v) )  // check perpendicularity
        {   std::cout << "No, not a rectangle." << std::endl;
            return 1; }
        u = v; }
    std::cout << "Yes, a rectangle." << std::endl;
    return 0; }
```
Optional and Variable-Length Arguments

- Functions with Optional Arguments

```cpp
class Time {
    public:
        Time() { }  
        Time(int hr, int min, int sec = 0, char ap = ‘A’); /* only once */
    }

    Time t1(2, 30, 0, ‘P’);
    Time t2(9, 15);
    Time t3(6, 15, 30);
    Time t4;
    Time::Time(int, int = 0, int, char); /* assume all right arg’s optional */
    Time::Time(int hr, int min, int sec /* = 0 */, char ap /* = ‘A’ */);
}
```

- Overloading Functions
  - Distinguishable Signatures
  - Function Call Resolution

```cpp
int power (int a, int n);
double power (double a, int n);
Fraction power (Fraction a, int n); /* not a member function */
```

- Friend of a class

```cpp
friend Fraction operator - (int i, const Fraction & r)  
{ return Fraction(i*r.denom – r.num, r.denom); }  
friend Fraction operator -(const Fraction & r, int i)  
{ return Fraction(r.num - i*r.denom, r.denom); }  

- Why not implement operator - as a member function?
- All member functions of a class X must be friends
friend class X;
```
Inheritance

- Reuse of existing code
- Adaptation of programs to work in similar but different situations
- Extraction of commonalities from different classes
- Organization of objects into hierarchies

### Basic Concepts and Syntax

- Private Members in Inheritance
- Adjusting Access
- Name Hiding
- Indirect Inheritance

```cpp
class Film {
public:
    void store_title( const string& t ) { title = t; }
    void store_title( const char* t ) { title = t; }
    void store_director( const string& d ) { director = d; }
    void store_director( const char* d ) { director = d; }
    void store_time( int t ) { time = t; }
    void store_quality( int q ) { quality = q; }
private:
    string title; string director;
    int time; // in minutes
    int quality; // 0 (bad) to 4 (tops)
};

class DirectorCut : public Film {
public:
    void store_rev_time( int t ) { rev_time = t; }
    void store_changes( const string& s ) { changes = s; }
    void store_changes( const char* s ) { changes = s; }
private:
    int rev_time; string changes;
};
class ForeignFilm : public Film {
public:
    void store_language( const string& l ) { language = l; }
    void store_language( const char* l ) { language = l; }
private:
    string language;
};
```
Constructors & Destructor under Inheritance

- **Derived Class Constructors Rules**
  - If DC has constructors but BC has no constructors, then the appropriate DC constructor executes automatically whenever a DC object is created.
  - If DC has no constructors but BC has constructors, then BC must have a default constructor so that it can execute automatically whenever a DC object is created.
  - If DC has constructors but BC has a default constructor, then BC’s default constructor executes automatically whenever a DC object is created unless the appropriate DC constructor explicitly invokes, in its initialization section, some other BC constructor.
  - If DC and BC have constructors but BC has no default constructor, then each DC constructor must explicitly invoke, in its initialization section, a BC constructor, which then executes when a DC object is created.

- **Destructor under Inheritance**

```cpp
class Sequence {
public:
  bool addS( int, const string& );   bool del( int );
  void output() const;

  Sequence( const char* );
  ~Sequence();

protected:
  enum { MaxStr = 50 };              string s[ MaxStr ];
  int last;

private:
  string filename;   ifstream in;
  ofstream out; 
};

bool Sequence::addS( int pos, const string& entry ) {
  if ( last == MaxStr – 1 || pos < 0 || pos > last + 1 )
    return false;
  for ( int i = last; i >= pos; i-- )
    s[ i + 1 ] = s[ i ];
  s[ pos ] = entry;   last++;   return true; }

bool Sequence::del( int pos ) {
  if ( pos < 0 || pos > last )  return false;
  for ( int i = pos; i < last; i++ )
    s[ i ] = s[ i + 1 ];
  last--;  return true; }

void Sequence::output() const {
  for ( int i = 0; i <= last; i++ )
    cout << i << " " << s[ i ] << "\n"; }
```
Sequence::Sequence( const char* fname ) {
    last = -1; filename = fname; in.open( fname );
    if ( !in ) return;
    while ( last < MaxStr - 1 && getline( in, s[ last + 1 ] ) )
        last++;
    in.close();
}

Sequence::~Sequence() {
    if ( filename == "" ) return;
    out.open( filename.c_str() );
    for ( int i = 0; i <= last; i++ )
        out << s[ i ] << '\n';
    out.close();
}

class SortedSeq : public Sequence {
public:
    bool addSS( const string& ); SortedSeq() {}
    SortedSeq( const char* );

protected:
    void sort();

private:
    using Sequence::addS; }

void SortedSeq::sort() {
    string temp; int i, j;
    for ( i = 0; i <= last - 1; i++ ) {
        temp = s[ i + 1 ];
        for ( j = i; j >= 0; j-- )
            if ( temp < s[ j ] )
                s[ j + 1 ] = s[ j ];
            else
                break;
        s[ j + 1 ] = temp;
    }

bool SortedSeq::addSS( const string& entry ) {
    int i;
    for ( i = 0; i <= last; i++ )
        if ( entry <= s[ i ] )
            break;
    return addS( i, entry );
}

SortedSeq::SortedSeq( const char* fname ) : Sequence( fname ) {
    sort();
}
class Account                       // class name
{public:
    Account(unsigned n, double b,  const string& ss);
    Account() {}                  // default constructor
    void deposit(double amt);      // deposit amt into this account
    bool withdraw(double amt);     // withdraw amt from this account
    double balance() const;        // balance inquiry
    unsigned id() const     {  return acct_no;  }
protected:
    unsigned  acct_no;             // account number
    string    ss;                 // owner ss no.
private:
    double    acct_bal; };  // current balance

class JointAccount : public Account      // derive from Account
{public:
    JointAccount(unsigned n, double b,  const string& owner,
               const string& jowner);
    JointAccount() {}                   // default constructor
private:
    string jss; }; // joint owner ss no.

class FreeChecking : public Account
{ public:
    FreeChecking(unsigned n, double b, const string& owner);
    FreeChecking() {}                   // default constructor
    void fee();                         // charge monthly fee
    bool withdraw(double amt);          // withdraw amt
    static void setMinbal(float m) { min_bal = m; }
    static void setFee(float f) { service_fee = f; }
    static float getMinbal() { return min_bal; }
    static float getFee() { return service_fee; }
private:
    static float min_bal;
    static float service_fee;
    bool free;   };}
Multiple Inheritance

- Inheritance and Access
- Virtual Base Class
- Protected Inheritance
  - Each public member in BC is protected in DC
  - Each protected member in BC is protected in DC
  - Each private member in BC is visible only in BC
- Private Inheritance
  - Each public member in BC is private in DC
  - Each protected member in BC is private in DC
  - Each private member in BC is visible only in BC

Derivation Principles

- Use public derivation when a derived object is a kind of base object
- Use private derivation when a derived object is not considered a kind of base object and the base class simply supply code to make the derived class easier to write
- Use protected derivation when you basically want a private base class but also want to make inherited accessible from further derived classes.

- Access Control under class derivation
  - Access to inherited members by appendant member and friends of the derived class independent of base-class designation
    - All nonprivate (public and protected) inherited members
    - Not allowed to private members
  - dependent of base-class designation
    - Public Base: Public as public and protected as protected
    - Protected Base: Nonprivate as protected
    - Private Base: No inherited members are accessible
## Derivation Principles

<table>
<thead>
<tr>
<th>Access from</th>
<th>Inherited Member</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Derived class &amp; its friends</td>
<td>Public</td>
<td>Protected</td>
<td>Private</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Subsequent derived class</td>
<td>If public/protected</td>
<td>If public/protected</td>
<td>No</td>
</tr>
<tr>
<td>Other outside functions</td>
<td>If public</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

- **Overriding**
  - By defining an appendant member of the same name

- **Exempting**
  - By including `using base_class::member`
  - It cannot be used to promote accessibility beyond the derived class

- **Type relations under Inheritance**
  - **Implicit Conversions**
    - Convert a derived object, reference or pointer to a corresponding public base type
    - Convert a public base offset pointer to member to a derived one
    - Inside the appendant part, these conversions are implicit
  - **Explicit Conversion**
    - The `static_cast` operator converts the base pointer to a derived pointer

- **Inheritance and overloading**
  - An appendant member function hides all inherited functions with the same name. The inherited functions are not visible but still accessible by explicit scoping

- **Assignment of objects**
  - **Built-in Object assignment**
    - Combined assignments have no built-in meaning
  - **Class-defined Object assignment**
Multiple Inheritance

- Multiple Inheritance
  - Multiple Base-class Scopes
  - Shared bases (virtual base)
- Operator overloading
  - At least one operand (argument) of the operator must be of user defined type
  - Operator functions cannot have default arguments
  - Brand new operator cannot be introduced
  - Operator precedence is built-in and fixed
  - =, [ ], (), and -> can only be overloaded only by member operator functions

```cpp
Fraction& Fraction::operator +=(const Fraction& fra)
{  *this = *this + fra;
   return *this;  }  //member operator function

Fraction& operator +=(Fraction& f1, const Fraction& f2)
{  f1 = f1 + f2;
   return f1;  }  //non-member operator function
```

```cpp
Cirbuf& Cirbuf::operator= (const Cirbuf& b)
{  if (&b == this) return(*this);
   head = 0;
   tail = length = b.length;
   if ( size != b.size )
   {  delete [] cb;
      size = b.size;
      cb = new char[size];  }
   for (int i=0 ; i < length ; i++)
      cb[i] = b.cb[(b.head + i) % size];
   return *this;  }
```
Polymorphism

- Requirements for C++ Polymorphism
  - There must be an inheritance hierarchy
  - The classes in the hierarchy must have a virtual method with the same signature
  - There must be either a pointer or a reference to a base class. It is used to invoke a virtual method
  - A constructor cannot be virtual. A destructor can be virtual
  - Only a non-static method may be virtual

```cpp
class Account
{
public:
    Account() {} // default constructor
    Account(unsigned id, double amt, char* s): acct_no (id), acct_bal(amt)
        { strcpy(ss, s); }
    virtual void deposit(double amt) { acct_bal += amt; }
    virtual bool withdraw(double amt)
        { if( amt > acct_bal ) return false;
         acct_bal -= amt; return true; }
    virtual unsigned id( ) const { return acct_no; }
    virtual double balance( ) const { return acct_bal; }
    virtual void display(ostream& out = cout ) const
        { out << "Account No: " << acct_no << endl << "Owner SS: " << ss << "Balance: " << acct_bal << endl; }
    virtual ~Account( ) { cout << "Account Destructor" << endl; }
protected:
    enum {SS_LEN = 12};
    unsigned acct_no; // account number
    char ss[SS_LEN]; // owner ss no.
private:
    double acct_bal; // current balance
};

class JointAccount : public virtual Account // derive from Account
{
public:
    JointAccount(unsigned n, double b, char* owner, char* jowner)
        : Account(n, b, owner) { strcpy(jss, jowner); }
    JointAccount( ) {} // default constructor
    void display(ostream& out=cout) const
        { out << "Account No: " << acct_no << endl << "Owner SS: " << ss << "Joint Owner SS: " << jss << "Balance: " << balance() << endl; }
protected:
    char jss[SS_LEN]; // joint owner ss no.
};
```
class FreeChecking : public virtual Account
{
public:
    FreeChecking(unsigned n, double b, char* owner) : Account(n, b, owner), free(true)
    {
        if (balance() < min_bal) free = false;
    }
    FreeChecking() {} // default constructor
    void fee(); // charge monthly fee
    bool withdraw(double amt)
    {
        bool ok = Account::withdraw(amt);
        if (ok)
            if (free && balance() < min_bal) free = false;
        return ok;
    }
    void display(ostream& out = cout) const
    {
        out << "Free Checking Account" << std::endl;
        Account::display();
    }
static float getMinbal(float m) { return min_bal; }
static float getFee(float f) { return service_fee; }
static void setMinbal(float m) { min_bal = m; }
static void setFee(float f) { service_fee = f; }
~FreeChecking() {}
};

private:
    static float min_bal;
    static float service_fee;
    bool free;
};

class JtFrChecking : public JointAccount, public FreeChecking
{
public:
    JtFrChecking() {} // default constructor
    JtFrChecking(unsigned n, double b, char* owner, char* jowner)
        : Account(n, b, owner), JointAccount(n, b, owner, jowner),
          FreeChecking(n, b, owner) {}
    void display(ostream& out = cout) const
    {
        out << "Free Checking Account" << std::endl;
        JointAccount::display();
    }
    bool transfer(float amt, Account& from, Account& to)
    {
        bool flag = from.withdraw(amt);
        if (flag) to.deposit(amt);
        return flag;
    }
    void show(Account* a[], int n)
    {
        for (int i=0; i < n; i++)
        {
            a[i]->display();
            cout << endl;
        }
    }
int main()
{
    JtFrChecking jfc(23456, 750.0, "025-72-5555", "024-88-3333");
    JointAccount ja(55123, 1600.0, "043-12-4444", "034-21-2222");
    FreeChecking fc(66432, 600.0, "098-02-1111");
    transfer(300, fc, jfc);
    transfer(200, ja, fc);
    Account* acnt[3];
    show(acnt, 3);
    return 0;
}
## Virtual Functions

- **Keys of plug Compatibility**
  - Interchangeable objects: a collection of similar objects
  - Uniform public interface
  - Polymorphic variables: public base pointer or reference
  - Dynamic access of interchangeable operations

- **Understanding Virtual Functions**
  - Once a base prototype is designated virtual, all derived-class functions with the same prototype are virtual
  - When a virtual function is invoked through a base-type pointer or reference, the actual run-time type of the object determines which virtual function to call
  - A pure virtual function is one that left undefined with the =0 notation. The presence of pure virtual functions makes a class abstract
  - A direct call of a member virtual function by another member function is considered to be invocation through the host pointer this
  - Only instance functions and destructors can be virtual. Constructors, new and delete cannot be virtual.
  - A pure virtual function can be left undefined but must be defined by a derived class before instances can be established
  - The access protection designation (public, protected, or private) of a virtual function is determined by the base class
  - Virtual functions can be implemented as inline functions
Object-Oriented Design

- Thee situations, a virtual function invocation is fixed at compile time
  - When a virtual function invoked through an object instead of a pointer or reference
  - When a virtual function call explicitly specifies the class scope
  - When a virtual function call by a constructor or destructor

- Planning Uniform Public Interfaces
- Interfaces
  - All members are public
  - All functions are pure virtual
  - Other members may include static const data fields, enum constants, and typedefs

- Inheritance Planning
  - Protected or private
  - Shared base or not
  - Member function overloaded
  - Pure virtual functions

- Virtual-Function Composition
- Object-Family Classes
- Decomposition Approaches
  - Procedural Decomposition
  - Object-Oriented Decomposition

- Object-Oriented Design Principles
  - Identifying Classes and Objects
  - External Behavior of Objects
  - Designing Objects
  - Relationships among Objects and Classes
  - Implementation

- Design Patterns
  - Creational patterns: object factory
  - Structure patterns: composite
  - Behavioral patterns: iterators

- The CRC Method
  - Class names
  - Responsibilities
  - Collaborators
Two main parts
- Common Language Runtime (CLR) manages execution of code and provides services
  - Common Type System defines the types of data that manages code can use.
  - Common Language Specification (CLS) defines features that every language for developing managed code must provide
  - Each language code compiles to MSIL, Microsoft Intermediate Language. Compiles to native code at runtime.
- .NET Framework Class Library provides a large and very useful set of types
  - Groups types into namespaces. Contains about 100 namespaces. For example,
    - System -- Contains fundamental types.
    - System.Drawing -- Provides graphics.
    - System.Windows.Forms -- For user interfaces in Windows-based applications

C# Features
- Object-oriented – used for business objects or system applications – internet applications. Goals include:
  - productivity and safety
  - power, expressiveness, and flexibility
- Combines rapid application development of Visual Basic with power of C++

How C# works
- Compiler translates C# source to MSIL code for a virtual machine, similar to a hardware processor
- During runtime the CLR use a Just-In-Time (JIT) compiler to translate the MSIL code to the instruction set of the processor
Event-Driven Programming

- **Paint Events**
  - The operating system manages its windows
  - User operations require windows to be redrawn.
  - Examples:
    - Restoring a window that was minimized
    - Uncovering a window that was covered

- **Graphics Mode**
  - Pixels (picture elements) are dots on the screen used to create graphic images. At a resolution of 1024 by 268, the screen displays 786,432 pixels.
  - (0,0) represents upper left corner of a window
  - First coordinate, x, increases to the right
  - Second coordinate, y, increases toward the bottom

- **Forms**
  - Form class represents a window. We inherit from Form so we can draw or add controls to the window.

```csharp
public class HelloCount : Form
    // HelloCount is a Form
Application.Run(new HelloCount())
    // start HelloCount waiting for Paint events
```

- **Event handlers**
  - Event handler code called when an event occurs.
  - When a Form needs to be redrawn, the system call the OnPaint method

```csharp
Protected override void OnPaint(PaintEventArgs e)
    Graphics g = e.Graphics;
    // provides a Graphics object with drawing methods
```

- **Draw a line**
  - Use Graphics g
  - g.DrawLine(black, x1, y1, x2, y2);
  - (x1, y1) and (x2, y2) are the endpoints of the line.
  - Pen black = new Pen(Color.black, 3);
  - // line will be three pixels thick
Event-Driven Programming (2)

- **Draw a rectangle**
  - g.DrawRectangle(Pen p, int x, int y, int w, int h);
  - (x, y) is the upper-left corner, w is the width, h is the length
  - To fill the rectangle use the FillRectangle method and a brush
  - g.FillRectangle(Brushes.Red, 150, 20, 100, 50);

- **Draw (or fill) an Ellipse**
  - Specify the rectangle that bounds the oval
  - g.DrawEllipse(Pen p, int x, int y, int w, int h);
  - For a circle make w and h equal.
  - g.DrawEllipse(Brush b, int x, int y, int w, int h);

- **Draw an arc**
  - Draws a portion of an ellipse.
  - Two extra arguments
    - int startAngle // In degrees
    - int sweep // size of the arc in degrees
  - 360 degrees in a complete circle, measured counterclockwise from 3 o’clock
  - g.drawArc(black, 50, 50, 200, 100, 45, 90);

- **Fonts**
  - new Font(String name, int size, FontStyle style)
  - name can be physical font such as “Arial” or logical font such as “FontFamily.GenericSerif”
  - style values -- FontStyle.Bold
  - Also Plain, Italic, Regular, Strikeout, Underline
  - size in points with 72 point per inch
  - Font f = new Font(“Arial”, 24, FontStyle.Bold);

- **Centering a string**
  - To center “Arial” horizontally draw at
    - (Width - w)/2 for the x-position, where
  - int w = (int)g.MeasureString(“Arial”, f).Width;
  - and Width is the width of the Form
Event-Driven Programming (3)

- **Color**
  - Color class in System.Drawing namespace
  - Many predefined colors, Color.blue, Color.pink, Color.DarkOrchid, ...
  - Color.FromArgb(int red, int green, int blue)
    - each varies from 0 to 255
    - 0,0,0 is black, 255,255,255 is white
    - 255,0,0 is red 0,255,0 is green

- **Using the Mouse**
  - MouseDown event -- user presses the mouse
    - handle in OnMouseDown method
  - protected override void OnMouseDown(MouseEventArgs e)
  - MouseUp event – user releases the mouse
    - handle in OnMouseUp method
  - e.X and e.Y give the position of the mouse

- **Using Keys**
  - KeyDown and KeyUp events represent physical pressing and releasing of a key
  - Handle in OnKeyDown and OnKeyUp methods
  - PaintEventArgs e, e.KeyCode – physical key
  - Constants Keys.Left, Keys.Right for arrow keys
  - KeyPress event represents the character typed
  - KeyPressEventArgs e, e.KeyChar – character, may have taken more than one keystroke ex. G

User Interfaces

- **Controls**
  - A control is a component with a visual representation
  - A Form is a Control that can contain other controls
  - Controls introduced in this chapter include Button, Label, TextBox, RadioButton, ComboBox, ListBox, PictureBox, and CheckBox
User Interfaces

- Configuring a Form
  - .NET Framework – write code in constructor
  - (using Visual Studio .NET – change properties, drag controls)
    ```csharp
    Text = "Button Press"; // text in form’s title
    Size = new Size(300, 200); // size of form
    Controls.Add(print); // add a button
    Controls.Add(message); // add a label
    print.Text = "Print"; // set button label
    print.Location = new Point(20, 30); // place button
    message.Text = "Message goes here"; // label
    Message.Size = new Size (message.PreferredWidth, message.PreferredHeight);
    // make label big enough to hold message
    
    (Using Visual Studio .NET, set properties to generate this code automatically)
    ```

- Events
  - A class may contain an event that allows it to provide notifications.
  - A Button has a Click event that will occur when the user click the button.
  - To make the button work, we register an EventHandler delegate with it.
  - When the user clicks the button, the code in the registered event handler will be called

- Handling a Click event
  ```csharp
  private Button print = new Button() // the control
  print.Click += new EventHandler(Print_Click);
  // register Print_Click method, which must be an EventHandler
  // type with two parameters and no return value
  protected void Print_Click(Object sender, EventArgs e){
    message.Text = "Hi there";
  } // set message label when user clicks button
  ```
The .NET Button code
- Declares a Click event (inherited from Control).
- Associates an EventHandler delegate as the type of method needed to handle a Click event.
- Raises the Click event when the user clicks the button, causing calls to all registered delegates.
- Declares a Print_Handler method for the Print button that fits the definition of the EventHandler delegate.
- Registers Print_Handler as a new delegate to handle the Click event for the Print button.

```csharp
/* A print button prints a message. A clear button erases the message. We use a Label to hold the message. */
using System;
using System.Drawing;
using System.Windows.Forms;
public class ButtonPress : Form { // Create the controls
    private Button print = new Button();
    private Button clear = new Button();
    private Label message = new Label();
    public ButtonPress() { // Captions
        Text = "Button Press";
        print.Text = "Print";
        clear.Text = "Clear";
        message.Text = "Message goes here";
        Size = new Size(300,200); // Sizes
        message.Size = new Size (message.PreferredWidth, message.PreferredHeight);
        print.Location = new Point(20,30); // Locations
        clear.Location = new Point(30 + print.Width, 30);
        message.Location = new Point(40 + print.Width + clear.Width, 30);
        Controls.Add(print); // Add them to the form
        Controls.Add(clear);
        Controls.Add(message);
        // Tell the Click events which methods to call
        print.Click += new EventHandler(Print_Click);
        clear.Click += new EventHandler(Clear_Click); }
    protected void Print_Click(Object sender, EventArgs e) { message.Text = "Hi there"; }
    protected void Clear_Click(Object sender, EventArgs e) { message.Text = ""; }
    static void Main() { Application.Run( new ButtonPress() ); }}
```
TextBox, RadioButton and ComboBox

- **Delegates**
  - A delegate defines a class that encapsulates one or more methods.
  - To make a button work we register an EventHandler delegate that .NET has predefined as
  - public delegate void EventHandler(Object sender, EventArgs e)
  - Print_Click must have these same parameters and return type.

- **TextBox**
  - A TextBox lets the user enter data.
  - By default it allows a single line.
  - Clicking EnterPrice adds value in text box to sum.
  - Clicking Average displays the average.

- **RadioButton and ComboBox**
  - All radio buttons in a Form belong to one group by default. Selecting one deselects the others.
  - A ComboBox contains a list of items, showing one, and allowing the user to pop up the rest.
  - Handle the button Click event. When the user clicks the button, we determine which radio button is check and which item is selected in the combo box.
  - Respond to radio buttons and combo box immediately.
  - Selecting a RadioButton generates a CheckedChanged event.
  - Selecting a ComboBox item generates a SelectedIndexChanged event.

- **StringBuilder**
  - A String cannot be changed.
  - Repeated concatenation causes the allocation of new String objects.
  - Using StringBuilder is more efficient.

```
StringBuilder s = new StringBuilder(“The “);
s.Append(animal1);
s.Append(“ and “);
```
/** Uses a combo box to choose a color, and two radio buttons to choose a shape. Uses a draw button to request drawing the selected shape in the selected color. */

using System;
using System.Drawing;
using System.Windows.Forms;
public class Select : Form {
    private Button draw = new Button(); // Create controls
    private RadioButton square = new RadioButton();
    private RadioButton circle = new RadioButton();
    private ComboBox color = new ComboBox();
    private Color c = Color.Yellow;
    public Select() {
        Text = "Select"; // Set captions
        draw.Text = "Draw";
        color.Text = "Choose a color";
        square.Text = "Square";
        circle.Text = "Circle";
        Size = new Size(500,250); // Set size
        int w = 20; // Set locations
        draw.Location = new Point(20,30);
        square.Location = new Point(w += 10 + draw.Width, 30);
        circle.Location = new Point(w += 10 + square.Width, 30);
        color.Location = new Point(w += 10 + circle.Width, 30);
        color.Items.Add("Red"); // Add items to combo box
        color.Items.Add("Green");
        color.Items.Add("Blue");
        Controls.Add(draw); // Add controls to form
        Controls.Add(square);
        Controls.Add(circle);
        Controls.Add(color);
        draw.Click += new EventHandler(Draw_Click);  } // Register event handler
    // Display chosen shape in selected color
    protected override void OnPaint(PaintEventArgs e)   {
        Graphics g = e.Graphics;
        Brush brush = new SolidBrush(c);
        if (square.Checked) g.FillRectangle(brush,100,100,100,100);
        else g.FillEllipse(brush,100,100,100,100);
        base.OnPaint( e );  }
    // Handle button click
    protected void Draw_Click(Object sender, EventArgs e) {
        if (color.SelectedItem.ToString() == "Red") c = Color.Red;
        else if (color.SelectedItem.ToString() == "Green") c = Color.Green;
        else c = Color.Blue;
        Invalidate();  }
    static void Main() {    Application.Run(new Select());  }
**PictureBox**

- Use PictureBox to display an image
- private PictureBox photo = new PictureBox();
- photo.Image = Image.FromFile(“gittleman.gif”);

**ListBox**

- Displays a list of items
- SelectionMode.One -- choose one item
- SelectionMode.MultiSimple – select multiple items
- Selecting an item generates a SelectedIndexChanged event
- SelectedItem property -- returns Object selected
- SelectedItems returns a collection of items selected. Use array notation to access each

**CheckBox**

- Can select more than one CheckBox, or none
- A user selection generates a CheckedChanged event
- Register an EventHandler delegate
- author.CheckedChanged
- += new EventHandler(Photo_Changed);
- Put event handling code in Photo_Changed method

**Using Visual Studio .NET**

- Create a C# Windows Application project
- Click on a control in Toolbox and click on Form to position the control
- Visual Studio .NET generates all the initialization code in the InitializeComponent method.
- Double-click on a control to bring up a code template for its event handler. Fill in code.
Handle unexpected errors gracefully.

An exception signals that a condition such as an error has occurred. We throw an exception as a signal, and catch it to handle it and take appropriate action.

Many predefined exception classes.

try-catch

We use predefined exceptions. The system throws an exception when the error condition occurs.

Put code that may throw an exception in a try-catch block

try {
   // code that may fail
} catch(Exception e) { // handling code }

Some exceptions

IndexOutOfRangeException
   – using an array index that is out of range
   – int[] a = {3,4,5}; a[5] causes an exception

FormatException improper format for the type expected
   – int.Parse("3.1459") fails

/* Shows the use of the exception object to obtain the sequence of method calls that culminated in the throwing of the index out of range exception. */
using System;
public class TryExceptionTrace{
   public static int GetAndSetValue(String value) {
      int[] anArray = {5,6,7};
      int index = int.Parse(value);
      return anArray[index];
   }
   public static void Main(String[] args) {
      int value;
      try {
         value = GetAndSetValue(args[0]);
         Console.WriteLine("Execution does not get here if index is bad");
      } catch (IndexOutOfRangeException e) { Console.WriteLine(e); }
      Console.WriteLine("This is the end of the program");
   }
}

System.IndexOutOfRangeException: Index was outside the bounds of the array.
at TryExceptionTrace.GetAndSetValue(String value)
at TryExceptionTrace.Main(String[] args)
This is the end of the program
Menus and Dialogs

/* Illustrates some menu and dialog features. */
using System;
using System.Drawing;
using System.IO;
using System.Windows.Forms;
public class MenuDialog : Form {

    // Create control
    TextBox text = new TextBox();
    public MenuDialog() {
        Size = new Size(500,200); // Configure form
        Text = "Menus and Dialogs";
        text.Size = new Size(450,120); // Configure text box
        text.Multiline = true;
        text.ScrollBars = ScrollBars.Both;
        text.WordWrap = false;
        text.Location = new Point(20,20);
        MenuItem fileMenu = new MenuItem("File"); // Configure file menu
        MenuItem open = new MenuItem("Open");
        open.Shortcut = Shortcut.CtrlO;
        MenuItem save = new MenuItem("Save");
        save.Shortcut = Shortcut.CtrlS;
        fileMenu.MenuItems.Add(open);
        fileMenu.MenuItems.Add(save); // Configure feedback menu
        MenuItem feedbackMenu = new MenuItem("Feedback");
        MenuItem message = new MenuItem("Message");
        message.Shortcut = Shortcut.CtrlM;
        feedbackMenu.MenuItems.Add(message); // Configure format menu
        MenuItem formatMenu = new MenuItem("Format");
        MenuItem font = new MenuItem("Font");
        font.Shortcut = Shortcut.CtrlF;
        formatMenu.MenuItems.Add(font);
        MainMenu bar = new MainMenu(); // Configure main menu
        Menu = bar;
        bar.MenuItems.Add(fileMenu);
        bar.MenuItems.Add(feedbackMenu);
        bar.MenuItems.Add(formatMenu);
        Controls.Add(text); // Add control to form
        open.Click += new EventHandler(Open_Click); // Register event handlers
        save.Click += new EventHandler(Save_Click);
        message.Click += new EventHandler(Message_Click);
        font.Click += new EventHandler(Font_Click);   }

    // Configure...
protected void Open_Click(Object sender, EventArgs e) {
    OpenFileDialog o = new OpenFileDialog();
    if(o.ShowDialog() == DialogResult.OK) {
        Stream file = o.OpenFile();
        StreamReader reader = new StreamReader(file);
        char[] data = new char[file.Length];
        reader.ReadBlock(data, 0, (int)file.Length);
        text.Text = new String(data);
        reader.Close();    }
}

protected void Save_Click(Object sender, EventArgs e) {
    SaveFileDialog s = new SaveFileDialog();
    if(s.ShowDialog() == DialogResult.OK) {
        StreamWriter writer = new StreamWriter(s.OpenFile());
        writer.Write(text.Text);
        writer.Close();    }
}

protected void Message_Click(Object sender, EventArgs e) {
    MessageBox.Show("You clicked the Message menu", "My message");
}

protected void Font_Click(Object sender, EventArgs e) {
    FontDialog f = new FontDialog();
    if(f.ShowDialog() == DialogResult.OK)
        text.Font = f.Font;
}

public static void Main() { Application.Run(new MenuDialog()); }
Networking

- **Protocols and the URL**
  - Protocols specify message formats for communicating computers
  - Uniform Resource Locator (URL) identifies a remote site
  - Four parts: protocol name, host address, port, path to resource

- **Some Protocols**
  - HTTP – Hypertext Transfer Protocol - browsers talk to web servers – port 80 default
  - SMTP – Simple Mail Transfer Protocol - send mail, uses port 25
  - POP3 – Post Office Protocol version 3 - read mail, uses port 110
  - Other protocols include FTP and Telnet

- **An HTTP request**
  - Client (browser) sends three-part request: Method used, Resource, & Protocol version e.g. GET /index.html HTTP/1.0
  - Client may send request headers with form Name : Value, e.g. Accept: text/plain
  - Client sends blank line to signal header end
  - Client may send data after blank line

```
GET /~artg/TryURL.cs HTTP/1.0
User-Agent: Java1.3.0
Host: www.cecs.csulb.edu
Accept: text/plain
Connection: Keep-alive
```

- **An HTTP response**
  - Server sends status line of form: HTTP version, Status code & Reason, e.g. HTTP/1.0 200 OK
  - Server may send response headers of form: Name : Value, e.g. Content-type: text/html
  - Server sends blank line to signal end of headers
  - Server sends data

```
Status line:
HTTP/1.0 200 Document follows
Response headers:
Date: Sat, 03 Aug 2002 16:50:44 GMT
Server: Apache/1.3.9 (Unix) PHP/4.0.6 mod_perl/1.21
Last-Modified: Sat, 03 Aug 2002 12:40:05 GMT
ETag: "10f16c-375-3d4bcf25"
Accept-Ranges: bytes
Content-Length: 885
Content-Type: text/plain
```
Making a connection

- WebClient class encapsulates several protocols, and handles the communication for a request
- WebRequest and WebResponse encapsulate several protocols, and handle the communication. In addition they allow us to customize and get connection info
- TcpClient makes a raw connection. Protocols must be explicit.

- Using WebClient

```csharp
WebClient client = new WebClient();
client.BaseAddress = args[0];
client.DownloadFile(args[1], args[2]);
StreamReader input = new StreamReader(client.OpenRead(args[1]));
Console.WriteLine(input.ReadToEnd());
WebHeaderCollection header = client.ResponseHeaders;
for (int i = 0; i < header.Count; i++)
    Console.WriteLine("   {0} : {1}", header.GetKey(i), header[i]);
```

- HttpWebRequest and HttpWebResponse

```csharp
HttpWebRequest request = (HttpWebRequest) WebRequest.Create("some url");
HttpWebResponse response = (HttpWebResponse)request.GetResponse();
request.Accept = "text/plain";  // set header
WebHeaderCollection header = request.Headers;
header.GetKey(i) is key, header[i] is value
response.ContentLength;  // a response header
```

- POP3 Example

```
Server:   +OK POP3 server ready       // server sends welcome
Client:     USER username            // client sends user's name
Server:    +OK                                             // server responds OK
Client:     PASS password           // client sends the password
Server:    +OK 23 messages 3040 octets    // server sends message info
Client:     RETR 23                  // asks for message 23
Server:     text of message 23, ending with a '.' alone on a line
Client:     QUIT
```

- TcpListener and TcpClient

```
TcpListener server = new TcpListener(5678);          // server listens on port 5678
TcpClient client = server.AcceptTcpClient();
// server waits for a connection from client
NetworkStream stream = client.GetStream(); // communicates with client
```
For ReverseServer and ReverseClient we create our own protocol.
To use HTTP a client must send request, request headers, blank line and data
To use HTTP a server must send response, response headers, blank line, and data

Threaded Web Server
Handles clients concurrently
Pass each client to a new thread
Create the input and output streams in the thread
Put the response in the Run method

VerySimpleBrowser

```csharp
/* Connects to a web server to download a text file. */
* Exercises suggest extensions to handle other file types. */
using System;
using System.Text;
using System.IO;
using System.Net.Sockets;
public class VerySimpleBrowser {
    public static void Main(String [] args) {
        TcpClient client = new TcpClient(args[0], int.Parse(args[1]));
        NetworkStream stream = client.GetStream();
        byte[] send = Encoding.ASCII.GetBytes
            ("GET " + args[2] + " HTTP/1.0 \r\n");
        stream.Write(send, 0, send.Length);
        send = Encoding.ASCII.GetBytes
            ("Host: " + args[0] + ':' + args[1] + "\r\n\r\n");
        stream.Write(send, 0, send.Length);
        int size = (int)client.ReceiveBufferSize;
        byte[] bytes = new byte[size];
        StreamReader reader = new StreamReader(stream);
        while(reader.ReadLine() != ""){
            String s;
            try{
                while((s = reader.ReadLine()) != null)
                    Console.WriteLine(s);
            }catch(IOException e) {} 
        }finally {
            reader.Close();
            stream.Close();
            client.Close();    }
    }
}
```
/* Serves a text file to an HTTP client submitting a GET request. Exercises suggest extensions to make the server more functional. */

using System;
using System.IO;
using System.Text;
using System.Net.Sockets;

public class VerySimpleWebServer {
    public static void Main(String[] args) {
        TcpListener server = new TcpListener(int.Parse(args[0]));
        server.Start();
        TcpClient client = server.AcceptTcpClient();
        NetworkStream stream = client.GetStream();
        StreamReader reader = new StreamReader(stream);
        String s = reader.ReadLine();
        String[] strings = s.Split();
        StreamWriter writer;
        if (strings[0] != "GET") {
            writer = new StreamWriter(stream);
            writer.WriteLine("HTTP/1.0 501 Not Implemented");
            writer.WriteLine();
        } else {
            String filename = strings[1];
            while(reader.ReadLine() != "");
            writer = new StreamWriter(stream);
            writer.WriteLine("HTTP/1.0 200 OK");
            writer.WriteLine("Content-type: text/plain");
            writer.WriteLine();
            StreamReader file = new StreamReader(filename);
            String z = file.ReadToEnd();
            writer.WriteLine(z);
            writer.Flush();
            writer.Close();
            file.Close();
        }
        client.Close();
        stream.Close();
        reader.Close();
        writer.Close();
        server.Stop();
    }
}