INHERITANCE + POLYMORPHISM:

```
class Employee {
    protected:
        string name;
        double pay;
    public:
        Employee() {
            name = "";
            pay = 0;
        }
        Employee(string empName, double payRate) {
            name = empName;
            pay = payRate;
        }
        string toString() {
            stringstream stm;
            stm << name << ": " << pay;
            return stm.str();
        }
        virtual double grossPay(int hours) {              // Virtual makes it polymorphic, so manager
            return pay * hours;                           // uses its own grossPay method
        }
};


class Manager : public Employee {                         // : Inherits Employee constructor and methods
    private:
       bool salaried;

    public:                                               // Constructor inherits from Employee(name, pay)
        Manager(string name, double pay, bool isSalaried): Employee(name, pay)
         {
             salaried = isSalaried;
         }
                                                          // With Virtual, this method will get used if
        virtual double grossPay(int hours) {              // grossPay is called for manager object
            if (salaried) {
                return pay;
            } else {
                return pay * hours;
            }
        }
        string toString() {
            stringstream stm;
            string salary;
            if (salaried) {
                salary = "Salaried";
            } else {
                salary = "Hourly";
            }
            stm << name << ": " << pay
                << ": " << salary << endl;
            return stm.str();
        }
};
```

ABSTRACT CLASS:

```
class Shape {                                       // Can't be instantiated, but instead is base model
        virtual void setLoc(int loc) = 0;           // Each method is virtual and abstract ( = 0 )
        virtual int getLoc() const = 0;             // This forces child classes to define these methods
        virtual void draw() const = 0;
};

class Circle : public Shape {                       // Circle inherits shape methods
    private:
        int location, radius;

    public:
        Circle(int loc, int r) {
            location = loc;
            radius = r;
        }
        virtual void setLoc(int loc){               // Must define virtual methods
            location = loc;
        }
        void setRadius(int r){                      // Note inherited and so not virtual
            radius = r;
        }
        virtual int getLoc()const{                  // Must define virtual methods
            return location;
        }
        int getRadius()const{                       // Note inherited and so not virtual
            return radius;
        }
        virtual void draw()const{                   // Must define virtual methods
            cout << "Drawing circle at: " << getLoc() <<
                    "\n With a radius of: " << getRadius() << endl;
        }
};

// Another simple example
class Animal {
    public:
        virtual void talk() = 0;                    // Virtual abstract method = 0
};

class Dog: public Animal {                          // Dog inherits Animal abstract method talk
    public:
        virtual void talk() {                       // Talk must be defined for Dog object
            cout << "Bow Wow!" << endl;
        }
};

class Cat: public Animal {                          // Cat inherits Animal abstract method talk
    public:
        virtual void talk() {                       // Talk must be defined for Cat object
            cout << "Meow!" << endl;
        }
};

int main() {
    Dog fido;
    Cat kitty;
    fido.talk();                                    // Output:     Bow Wow!
    kitty.talk();                                   // Output:     Meow!
    return 0;
}
```

GENERIC FUNCTION and ARGUMENT:

```
template <typename T>                                   // Using generic types

T max(T &arg1, T &arg2) {                               // Generic return type, Generic arguments referenced
    if(arg1 > arg2){
        return arg1;
    } else {
        return arg2;
    }
}

int main() {
    int a = 12;
    int b = 24;
    cout << max(a,b) << endl;                           // Display: 24
    double x =2.25;
    double y = .25;
    cout << max(x,y) << endl;                           // Display: 2.25
    string w1 = "apple";
    string w2 = "aardvark";
    cout << max(w1,w2) << endl;                         // Display: apple
    return 0;
}
```

GENERIC FUNCTION WITH MULTIPLE GENERIC TYPES:

```
template <typename T, typename U>                       // Two generic types

T max(T arg1, U arg2) {                                 // Generic class with two generic args
    return (arg1 > arg2) ? arg1 : arg2;                 // Conditional return
}                                       // MUST BE CAREFUL OF RETURN TYPE (Widen or Narrow conversion)

int main() {
    double num1 = 35.55;
    int num2 = 13;
    cout << "Max value is: " << max(num1, num2) << endl;
    return 0;                           // If num1 and num2 are reversed, you get narrowing conversion
}
```

GENERIC CLASS WITH MULTIPLE GENERIC ARGUMENTS:

```
template <class T, class U>                             // Generic template for two types
class CMap {
    private:
        vector<T> keys;                                 // First generic vector
        vector<U> values;                               // Second generic vector
    public:
        void insert(T key, U value){                    // Insert both types into each vector
            keys.push_back(key);
            values.push_back(value);
        }
        void get(int n){
            cout << keys[n] << ": " << values[n];       // Display both vector values depending on position
        }
};
int main() {
    CMap<string, int> grades;                           // Instantiate with two different types
    grades.insert("Jones", 88);                         // Insert two types at a time
    grades.insert("Smith", 90);
    grades.get(0);                                      // Display:  Jones: 88
    cout << endl;
    grades.get(1);                                      // Display:  Smith: 90
    return 0;
}
```

GENERIC CLASS:

```
template <typename T>                                   // Generic type T
class Stack {
    private:
        int datastore[100];
        int top;
    public:
        Stack(){
            top = -1;
        }
        void push(int num) {
            top++;
            datastore[top] = num;
        }
        T pop() {                                       // Generic function
            int val = datastore[top];                   // Store top number from stack
            datastore[top] = 0;                         // Clear out spot with zero for numerical type
            top--;
            return val;
        }
        T peek() {                                      // Generic function
            return datastore[top];
        }
};

template<>                                              // Generic class
class Stack<string> {                                   // Generic class defined using string type
    private:
        string datastore[100];                          // Same as with original, but with string type
        int top;
    public:
        Stack(){
            top = -1;
        }
        void push(string val) {                         // Push specific string type
            top++;
            datastore[top] = val;
        }
        string pop() {                                  // Specific string type pop
            string val = datastore[top];                // Store string type from top of stack
            datastore[top] = "";                        // Blank out that spot in stack
            top--;
            return val;
        }
        string peek() {                                 // Specific string type peek
            return datastore[top];
        }
};

int main() {
    Stack<double> numbers;                              // Stack with double as type
    numbers.push(12);
    numbers.push(22);
    cout << numbers.peek() << endl;                     // Display: 22
    numbers.pop();                                      // remove top of stack
    cout << numbers.peek() << endl;                     // Display: 12
    Stack<string> names;                                // Stack using the generic stack<string> class
    names.push("Mary");
    names.push("Joe");
    cout << names.peek() << endl;                       // Display: Joe
    return 0;
}
```

EXCEPTION HANDLING <stdexcept>:

```
// Simple try/catch error handle
const int DivideByZero = 1;
int main() {
    int numer = 12, denom = 0;
    try{                                                // Try to divide
        if(denom == 0)
            throw DivideByZero;                         // if denom is 0 throw error
        else
            cout << numer / denom;
    }
    catch(int e){                                       // Here the error is handled
        if (e == DivideByZero)
            cout << "Can't divide by zero\n";
    }
    return 0;
}

// Another example - using a class to handle errors
#include <stdexcept>                                    // need to include exception header
class DivideByZero : public runtime_error {             // inherit from exception header
    public:
        DivideByZero() :
            runtime_error("Divide by zero exception") {}  // What to display
};
double quotient(double numer, double denom) {
    if(denom == 0)
        throw DivideByZero();                           // detect the error within the class
    else
        return numer/denom;                             // otherwise return quotient value
}
int main() {
    double num1, num2, ratio;
    cout << "Enter a numerator: ";
    cin >> num1;
    cout << "Enter a denominator: ";
    cin >> num2;
    try {                                               // Try to divide
        ratio = quotient(num1, num2);
        cout << "Result: " << ratio << endl;
    }
    catch (DivideByZero &except) {                      // If an error is detected – reference exception
        cout << except.what() << endl;                  // Displays "Divide by zero exception"
    }
    // Only use this if you're not sure what error will be thrown.
    catch (...){                                        // Alternative catch will catch every error (...)
        cout << "Exception thrown and caught" << endl;
    }
    return 0;
}

// Another example - file input catching error
int main(){
    ifstream file;
    file.exceptions(ifstream::failbit |
                    ifstream::badbit);
    try {
        file.open("file.txt");
        while(!file.eof())
            cout << file.get();
    }
    catch(ifstream::failure e){
        cout << e.what() << endl;
        cout << "Error opening file." << endl;
        return 1;
    }
    file.close();
    return 0;
}
```

STREAM INPUT/OUTPUT <sstream>, <fstream>:

```
//  stringstream used for storing a string of characters to be converted later
#include <sstream>

class Person {
    private:
       string first, middle, last;
       int age;

    public:
       Person(string f, string m, string l, int ag) {
           first = f; middle = m; last = l; age = ag;
       }

       string ToString() {
           stringstream stm;                                    //  Create stringstream object
           stm << first << " " << middle << " " << last << " " << age;   //  Store all, including int into stm
           return stm.str();                                    //  return stm as a string to be displayed
       }
};

int main() {
    Person aPerson("George", "Fred", "Jones", 27);
    cout << aPerson.ToString() << endl;                         //  Output everything as a string
    return 0;
}

//  Another example, this time with multiple datatypes from a file
#include <sstream>
#include <fstream>

int main(){
    //  grades.txt has "100 90 80 70 60" on the first line.
    ifstream gradeFile;                                         //  Create file object
    stringstream grades;                                        //  Create stream object
    int grade;
    int total = 0;
    string line;

    gradeFile.open("D:\\Save\\Code\\Practice\\grades.txt");     //  Open file, store in file object
    getline(gradeFile, line);                                   //  Get first line in file object
    grades << line;                                             //  Store that line in stringstream object
    gradeFile.close();                                          //  Close file object

    for(int i = 0; i < 5; i++){                                 //  For each part of the stringstream...
        grades >> grade;                                        //  output it to grade
        total += grade;                                         //  for each grade, add it to total
    }
    double average = total / 5;
    cout << "Average: " << average << endl;
    return 0;
}
```

SEQUENTIAL CONTAINERS <List>:

```
//  Linked List.. Good for random access
#include <list>                                          //  Include link list header
void display(list<string> pnouns) {                      //  Pass list to function to display
    list<string>::iterator itr = pnouns.begin();         //  Create iterator object to start at beginning
    while(itr != pnouns.end()) {                         //  While list has stuff in it
        cout << *itr << endl;                            //  Display item at iterator's point
        itr++;                                           //  Increment iterator
    }
}
int main(){
    list<string> names;                                  //  Create list object
    names.push_back("Mary");
    names.push_back("Zach");                             //  Add some items using .push_back (back of list)
    names.push_back("Elizabeth");                        //  push_front adds items to front of list
    cout << "\nReversed:" << endl;
    names.reverse();                                     //  Reverse items in list
    display(names);
    cout << "\nSorted:" << endl;
    names.sort();                                        //  Sort items in list
    display(names);
    return 0;
}
```

ASSOCIATIVE CONTAINERS – <utility>, <map> - PAIR, MAPS:

```
//  Pairs can store two different datatypes
#include <utility>                                       //  <utility> for pair
int main() {
    pair<string, int> num1("Jones", 123);                //  pair<type, type> name(data,data)
    cout << num1.first << " : " << num1.second << endl;  //  name.first , name.second
    return 0;
}

//  Map uses pairs to store information
#include <map>                                           //  Include map
void display(map<string,int> num){                       //  display receives a map type
    map<string, int>::iterator itr = num.begin();        //  Iterator for map, set it to the beginning
    cout << "The List: \n";
    while(itr != num.end()){                             //  While iterator isn't at the end
        cout << itr->first << " : " << itr->second << endl;  //  Display first and second elements of map
        itr++;
    }
}

int main() {
    map<string, int> numbers;                            //  create our map of strings and ints
    numbers["Jones"] = 365;
    numbers["Smith"] = 467;                              //  Add elements
    numbers["Brown"] = 111;
    string name;
    cout << "Enter a name to find: ";
    cin >> name;

    if(numbers.find(name) != numbers.end()) {            //  Search all elements till end for user entered
        cout << name << " : " << numbers[name] << endl;  //  Display found name and number
    }
    else {
        cout << name << " not found!" << endl;           //  If not, display "'name' not found."
    }

    display(numbers);
    cout << "Number(before erase): " << numbers.size() << endl;  //  Display number of elements in map

    numbers.erase("Smith");                              //  Erase a particular element
    cout << "\nNumber(after erase): " << numbers.size() << endl;
    display(numbers);
    return 0;
}
```

```
//  Sets store unique elements following a specific order
#include <set>
int main(){
    set<string> words;                                      //  Set object created
    string word = "";

    do{
        cout << "Enter a word: ";
        cin >> word;
        words.insert(word);                                 //  Add words as long as they're unique
    } while (word != "q");
    set<string>::iterator itr = words.begin();              //  Set iterator

    while(itr != words.end()){                              //  Run till end of set
        cout << *itr << " ";                                //  Display all words stored in set
        itr++;
    }
    return 0;
}
```

```
//  Multimap store elements by combined key value and mapped value (used in binary search trees)
#include <map>                                              //  Need to include both <map> and <utility>
#include <utility>                                          //  to use map and pair
using namespace std;
int main() {
    multimap<string, string> numbers;                       //  multimap object
    numbers.insert(pair<string,string>("Jones","123"));
    numbers.insert(pair<string,string>("Smith","245"));
    numbers.insert(pair<string,string>("Brown","111"));
    numbers.insert(pair<string,string>("Jones","333"));     //  Adding elements to multimap
    numbers.insert(pair<string,string>("Green","834"));     //  several "Jones" added
    numbers.insert(pair<string,string>("Jones","627"));

    string searchName = "Jones";
    multimap<string, string>::iterator itr =                //  Set itr to first Jones
            numbers.find(searchName);
    multimap<string, string>::iterator last =               //  Set last to final Jones
            numbers.upper_bound(searchName);

    for(;itr != last; itr++){                               //  while itr isn't pointed at last
        cout << itr->first << " : " << itr->second << endl;     //  Display first and second value of each
    }
    return 0;
}
```

NAMESPACES:

```
namespace firstNums {                                       //  namespace firstNums created with data
    int num1 = 10;
    int num2 = 12;
}
namespace secondNums {                                      //  namespace secondNums create with data
    int num1 = 1;
    int num2 = 2;
}
int main(){
    cout << "num1 in firstNums: " << firstNums::num1 << endl;     //  call num1 out of firstNums with scope op
    cout << "num1 in secondNums: " << secondNums::num1 << endl;   //  call num1 out of secondNums with scope op

    {                                                       //  Note blocking to insure locality
        using namespace firstNums;                          //  locally use firstNums variables
        cout << "num1 in firstNums: " << num1 << endl;      //  num1 from firstNums is in use
    }                                                       //  close block to insure locality

    {
        using namespace secondNums;                         //  ability to use secondNums num1
        cout << "num1 in secondNums: " << num1 << endl;
    }
    return 0;
}
```

```
//  Advanced use of namespace with functions and classes
namespace minMax {                                          //  Incorporating a function inside a namespace
    int min(int num1, int num2){
        if (num1 < num2) {
            return num1;
        }
        else {
            return num2;
        }
    }
    int max(int num1, int num2) {
        if (num1 > num2) {
            return num1;
        }
        else {
            return num2;
        }
    }
}
namespace People {                                          //  Incorporating a class inside a namespace
    class Person {
        private:
           string name;
           string sex;
        public:
           Person(string n, string s) {
               name = n;
               sex = s;
           }
           string get() {
               return name + ", " + sex;
           }
    };
}
int main()
{
    using namespace minMax;
    using namespace People;
    int a,b;
    cout << "Enter a number: ";
    cin >> a;
    cout << "Enter another number: ";
    cin >> b;
    cout << min(a,b) << endl;                               //  use function from minMax namespace
    cout << max(a,b) << endl;                               //  use function from minMax namespace
    cout << endl;
    Person you("Jane Doe", "F");                            //  instantiate People object
    cout << you.get() << endl;                              //  use the get() function from People object
    return 0;
}
```

STRING CLASS:

```
//  Different ways of using a string
#include <string>
int main()
{
    string str0;                                            //  string with no value
    string str1 = "";                                       //  empty string
    string str2(str1);                                      //  Contents of str1 are now also in str2
    string str3("a string");                                //  str3 = "a string"
    string str4(10,'*');                                    //  str4 = "**********"
    string str5 = "hello";                                  //  str5 = "hello"
    string str6 = "world";                                  //  str6 = "world"
    string str7 = str5 + " " + str6 + "!";                  //  str7 = "hello world!" note the space added
    string str8 = str5 + ", ";                              //  str8 = "hello, "
    string str9 = "hello" + ", " + str6;                    //  str9 = error cant concatenate two literals
    return 0;
}
```

```
//  Compare method
#include <string>
int main()                                          //  1 if greater than, 0 if equal, -1 if less than
{
    string s1 = "clean";
    string s2 = "clear";
    cout << s1.compare(s2) << endl;                 //   returns -1 since because s1 is less than s2
    return 0;
}

//   find, rfind, find_first_of
#include <string>
int main()
{
    string s1 = "a pin in a haystack pin";
    int pos = s1.find("pin");                       //   first occurrence
    int pos = s1.rfind("pin");                      //   last occurrence

    string numbers = "0123456789";
    string identifier = "name";
    int pos = identifier.find_first_of(numbers);    //   any occurrence of numbers in identifier
}

//  substr method
int main()
{
    string s1 = "a needle in a haystack";
    string word = "needle";
    int pos = s1.find(word);                        //   find word string in s1 string, store pos
    string s2 = s1.substr(pos, word.length());      //   make s2 = substr from pos to length of word
    string s3 = s1.substr(pos+word.length()+1);     //   make s3 = starting past word+space on
    cout << s3 << endl;                             //   s3 = ”in a haystack"
    s1.replace(pos, word.length(), "pin");          //   replace ”needle” with ”pin”
    return 0;
}
```