1. An *Abstract Data Type* (ADT) is a programmer defined data type that specifies: (1) values that can be stored, and (2) operations that can be done on the values. The user of an abstract data type does not need to know the implementation of the data type, *e.g.*, how the data is stored or how its operations are performed. C++ *classes* define the blueprints for ADTs, and we can construct as many *instances* of the class, called *objects*. For example, in addition to C-style (null terminated character arrays), C++ has a `string` class. You can define a string object by calling any constructor for a string.

Definition	**Meaning**
`string name;`	defines an empty string object
`string myname("Chris");`	defines a string and initializes it
`string yourname(myname);`	defines a string and initializes it
`string aname(myname, 3);`	defines a string and initializes it with first 3 characters of `myname`
`string verb(myname,3,2);`	defines a string and initializes it with 2 characters from `myname` starting at position 3
`string noname('A', 5);`	defines string and initializes it to 5 `'A'`s

Unlike traditional c-strings, which are mere sequences of characters in a memory array, C++ string objects belong to a class with many built-in features to operate with strings in a more intuitive.

OPERATOR	MEANING
`>>`	extracts characters from stream up to whitespace, insert into string
`<<`	inserts string into stream
`=`	assigns string on right to string object on left
`+=`	appends string on right to end of contents on left
`+`	concatenates two strings
`[]`	references character in string using array notation
`>, >=, <, <=, ==, !=`	relational operators for string comparison. Return `true` or `false`

Details about the following member functions can be found at:

http://www.cplusplus.com/reference/string/string/

Capacity Related Member Functions		
size or length	`cout << myStr.size();` `cout << myStr.length();`	Return length of string
max_size	`cout << myStr.max_size();`	Return maximum # character any string can hold
resize	`myStr.resize(15);` `myStr.resize(50,'*');`	Resize string. Truncate the string if is shrinking in size, or paddle with the specified character (default '\0').
capacity	`cout << myStr.capacity();`	Return size of allocated storage
reserve	`myStr.reserve(15);`	Request a change in capacity to at least specified size
clear	`myStr.clear();`	Clear string so that it is empty
empty	`if (myStr.empty()) {`	Test if string is empty

String Modifier Member Functions		
append	myStr.append(anotherStr);	Append to string
push_back	myStr.push_back('*');	Append character to string
insert	myStr.insert(5, anotherStr);	Insert into string at a specific location
erase	myStr.erase(startPos, len);	Erase len characters from string at specific startPos
replace	myStr.replace(startPos, len, anotherStr)	Replace part of string by some other specific content
copy	myStr.copy(charArray, startPos, len);	Copy sequence of characters from string to char array
swap	myStr.swap(anotherStr);	Swap contents with another string

String operations		
c_str	`myCStr = myStr.c_str();`	Returns pointer to C string equivalent
data	`myCharArray = myStr.data();`	Returns pointer to char array containing myStr
find	`foundPos=myStr.find(otherStr,startPos)`	Returns position of first occurrence of a string (c-string, or char) on or after startPos
rfind	`foundPos=myStr.rfind(otherStr,startPos)`	Returns position of last occurrence of a string (c-string, or char) on or after startPos
find_first_not_of	`pos=myStr.find_first_not_of("aeiou", startPos)`	Returns position of first char NOT in the specific string (c-string, or char) on or after startPos
substr	`otherString = substr(startPos, len);`	Returns a string of len characters starting at startPos

Suppose we want to read a line of text containing tab separated data and split it up into an array of substrings:

```
// Demo of C++ string class usage
#include <iostream>
#include <string>
using namespace std;

const int SIZE = 100;

// prototypes
void splitLineByCharacter(string myLine, char delimiter, string substrings[],
                          int & substringCount);
int main() {
   int substringCount, index;
   string myLine;
   string substrings[SIZE];

   cout << "Enter your <Tab> separated data: ";
   getline(cin, myLine);   // reads a whole line of text including whitespace characters
   cout << "You entered address " << myLine << endl;

   splitLineByCharacter(myLine, '\t', substrings, substringCount);

   cout << "There are " << substringCount
                        << " <Tab> delimited strings.  The strings are:" << endl;
   for (index = 0; index < substringCount; index++) {
      cout << substrings[index] << endl;
   } // end for

} // end main
```

a) Which string member functions would be useful in writing the `splitLineByCharacter` function?

b) Write the code to implement the `splitLineByCharacter` function.

```
void splitLineByCharacter(string myLine, char delimiter, string substrings[],
                                                          int & substringCount) {
```

2. A *structure* is a C (and C++) construct that allows multiple variables of potentially different types to be grouped together. The general format for defining a structure is:

```
struct <structName>
{
   type1 field1;
   type2 field2;
   . . .
};  // NOTE the ';' after the definition
```

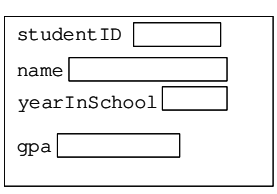
For example, we can define the template for a `Student` structure as:

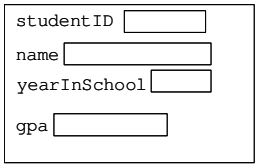
```
struct Student {
     int studentID;
     string name;
     short yearInSchool;
     double gpa;
}; // end Student struct
```

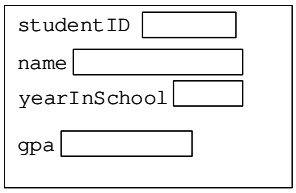
Like the recipe for a cake, we don't actually have any Student structures until we define variables by using the structure-name Student as a type name:

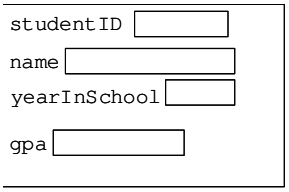
```
Student bill = {123456, "Bill", 3, 3.10};
Student sally, myClass[50];
```

bill: sally: myClass: 0 1 ....









:

The dot (.) operator is used to refer to members of struct variables:

```
cin >> sally.studentID;
getline(cin, sally.name);
sally.yearInSchool = 2;
sally.gpa = 3.75;
myClass[0].gpa = 4.0
```

a) Write code to display the information about student variable bill.

b) Unlike arrays, a structure variable definition does NOT create a pointer to a chunk of memory. Thus, if you write a `getStudent` function to interactively read information about a student, you have several options on returning the `Student` information back to the main:

- use a pass-by-reference Student parameter
- return the `Student` as the return type with the call from the main looking like:

```
Student myStudent;
myStudent = getStudent( );
```

- pass a pointer to a Student structure to the the function

Write this `getStudent` function that returns a `Student` as the return type.

c) A third approach would be to pass a pointer to a Student structure to the the function as:

```
Student * newStudentPtr;
newStudentPtr = new Student;        // dynamically allocate a Student record
getStudent(newStudentPtr);          // call by sending the pointer newStudentPtr
```

In the `getStudent` function definition, show how would you read a value for the `studentID` member?

```
void getStudent(Student * myStudentPtr) {
      cout << "Enter Student's ID: ";
      cin >>
      ...
}
```

d) Alternatively, if you have a pointer to a structure as in the previous `getStudent` function, you can use the "follow the pointer operator" `->` as shown below. How would you read a value for the `name` member?

```
void getStudent(Student * myStudentPtr) {
      cout << "Enter Student's ID: ";
      cin >> myStudentPtr->studentID;

      cout << "Enter Student's name: ";
      cin >>
      ...
}
```

3. An *enumerated type* in C++ is a data type whose legal values are a set of named constant integers. An example would be:

```
      enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY };
```

A variable of type Day can only be assigned one of these constant values.

```
int myInt, weekDay;
Day today, tomorrow;

today = MONDAY;
```

In C++ enumerated types are inplemented using integer. In the type Day, Sunday is a named constant integer with the value 0, Monday is a named constant integer with the value 1, etc.

a) The assignment statements:

```
      today = MONDAY;
      myInt = today;   // myInt gets the integer value 1
```

are both legal, but

```
      myInt = 1;
      today = myInt;
```

causes an error. Why?

b) Explain the output of the following code.

```
#include <iostream>
using namespace std;

enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY };

int main() {
      int myInt, weekDay;
      double totalHours, hoursWorked[] = { 10, 5, 5, 9, 8, 9, 12 };
      Day today;

      today = MONDAY;
      cout << "today's value: " << today << endl;
      today = MONDAY;
      myInt = today;    // myInt gets the integer value 1
      cout << "myInt's value: " << myInt << endl;

      myInt = 1;
      //today = myInt;  CAUSES AN ERROR
      today = static_cast<Day>(myInt+1);
      cout << "today's value: " << today << endl;

      totalHours = 0;
      for (weekDay = MONDAY; weekDay <= FRIDAY; weekDay++) {
            totalHours = totalHours + hoursWorked[weekDay];
      } // end for
      cout << "Week Day total hours:  " << totalHours << endl;
      cout << "Week-End total hours:  " << hoursWorked[SATURDAY]+hoursWorked[SUNDAY] <<endl;
} // end main
```

```
today's value: 1
myInt's value: 1
today's value: 2
Week Day total hours:  36
Week-End total hours:  22
```

C++ *classes* define the blueprints for ADTs, and we can construct as many *instances* of the class, called *objects*. For example, C++ has a `string` class.

a) If you download the receipe for a cake from the Internet, how many cakes do you have?

b) How many cakes can you make from this receipe?

c) In object-oriented languages like C++, programmers define *classes* ("the receipes") and can make (*construct*) as many objects (*class instances*) as needed. Consider the following main program that uses a simple Die class. How many Die objects are used in this program?

```
/* File: TestDieMain.cpp to test the Die class */
#include "Die.h"
#include <iostream>
using namespace std;

int main() {
       Die die1 = Die();          // 6-sided die
       Die die2 = Die(8);  // 8-sided die

       cout << "die1: " << die1.getRoll() << " die2: "
     << die2 << endl;
       ++die1;
       cout << "++die1 " << die1 << endl;

       cout << "Rolls two dice 10 times: " << endl;
       for (int count=0; count < 10; count++) {
              die1.roll();
              die2.roll();
              cout << die1 << " " << die2;
              if (die1 == die2) {
                     cout << " dice are equal";
              } // end if
              cout << endl;
       } // end for
} // end main
```

```
die1: 5 die2: 7
++die1 6
Rolls two dice 10 times:
3 5
5 1
3 1
2 5
1 1 dice are equal
3 7
5 5 dice are equal
1 2
1 1 dice are equal
3 4
```

```
/***************************** Die.h ********************************
* Declaration of Die class
*****************************************************************/

#ifndef DIE_H
#define DIE_H
#include <iostream>
#include <iomanip>
using namespace std;

class Die {
       // These needed to be implemented as non-member functions since the left
       // operand is not a Die object.  We make them friend functions to allow
       // access to the private data members.
       friend ostream & operator<<( ostream &, const Die &);
       friend istream & operator>>( istream &, Die &);

private:
       int numberOfSides;
       int currentRoll;

public:
       Die(int sides = 6);  // default constructor with default sides of 6

       void roll();          // rolls the die

       int getRoll() const; // returns the value of the current roll

       bool operator==( const Die &) const;

       Die & operator++();

}; // end Die class

#endif
```

```
/*************************** Die.cpp ****************************
* Implementation of die class
*****************************************************************/
#include "Die.h"
#include <cstdlib>
#include <ctime>
#include <cassert>
#include <iostream>
using namespace std;

// constructs a die with the specified number of sides
Die::Die(int sides) { //
      assert(sides >= 1);
      srand( time(NULL) );     // initialize the random number generator
      numberOfSides = sides;
      currentRoll = rand() % sides + 1;
} // end Die constructor


void Die::roll() {          // rolls the die
      currentRoll = rand() % numberOfSides + 1;
} // end roll


int Die::getRoll() const {       // returns the value of the current roll
      return currentRoll;
} // end getRoll

bool Die::operator==( const Die & RHSDie) const{
      return (this->currentRoll == RHSDie.currentRoll);

} // end operator==


Die & Die::operator++( ) {
      if (currentRoll < numberOfSides) {
            currentRoll = currentRoll + 1;
      } // end if
      return *this;

} // end operator++

ostream & operator<<( ostream & output, const Die & die) {
      output << die.currentRoll;
      return output;  // enables "cout << a << b << c;"
} // end operator<<


istream & operator>>( istream & input, Die & die){
      input >> die.currentRoll;
      input >> die.numberOfSides;
      assert(die.currentRoll >= 1 && die.currentRoll <= die.numberOfSides);
      return input;
} // end operator>>
```