SHORT REVIEW OF CS TOPICS

RANDOM NUMBERS (2 MARKS)

Generating Random Numbers

The key function in generating random numbers is;

```c
int random (int n);
```

which generates a random number in the range of 0 to n-1. For example;

```c
y = random(100);
```

y will be in the range of 0 though 99.

The same set of random numbers will be generated if random() is used without randomize().

It would terrible if you were designing software for the gaming industry as everyone would know what numbers were going to come up.

Solution is to use randomize() before using random().

randomize() - used to seed the random number generator with a number which is developed from the system clock, which is always changing.

Typical Questions

In the following program, if the value of Guess entered by the user is 65, what will be the expected output(s) from the following options (i), (ii), (iii) and (iv)?

```c
#include <iostream.h>
#include <stdlib.h>
void main()
{
    int Guess;
    randomize();
    cin>>Guess;
    for (int I=1;I<=4;I++)
    {
        New=Guess+random(I);
        cout<<(char)New;
    }
}
```

(i) ABBC
(ii) ACBA
(iii)BCDA
(iv)CABD

While solving questions of this nature don't look at the options before listing all the possible outputs.

Just keep in mind random(int n) function generates numbers randomly from a set \{0-n-1\}, each number in this set has got equal probability to be generated.
List the output assuming generated random number is 0, thereafter for 1, 2, 3, ... and so on till n-1. After listing down all the possible outputs see if any of them is in options, if yes choose that one as your answer else write “None of these”.

Solution:

I varies from 1 to 4

New = 65 + random(1) // random(1)...0 to 0 Possible Output = A
New = 65 + random(2) // random(2)...0 to 1 Possible output = A or B
New = 65 + random(3) // random(3)...0 to 2 Possible output = A or B or C
New = 65 + random(4) // random(4)...0 to 3 Possible output = A or B or C or D

Now output could be any combination of four outputs

e.g. AAAA, ABCD, ABBC and so on

answer ABBC
INHERITANCE (4 MARKS)

Inheritance is a concept used in OOP to implement reusability & extensibility of classes.

Reusability: Reusing the code of existing classes to derive new classes from them.

Extensibility: Adding extra/new features to existing classes.

What are we going to inherit?

We inherit few or all characteristics and behaviour of existing classes.

(In c++ jargon characteristics is called data and behaviour is referred as method/function)

Benefit: Inheriting part of a program saves you countless hours of programming.

Also provides a way to limit access to attributes and behaviour.

Types of Inheritance: Simple, Multiple, Multilevel, Hybrid etc (Each enables a class to access attributes & behaviour of another class using slightly different techniques)

Types of visibility mode: Private, Public, Protected

Syntax of derived class:

```
derived class name is derived from Inheritance type: public or private base class name
```

class DerivedClass:[VisibilityMode] BaseClass
{
    // members of derived class
    // and they can access members of the base class
};

**Figure 14.2: Syntax of derived class declaration**

The following are the three possible styles of derivation:

1. class D: public B // public derivation
   {
       // members of D
   }

2. class D: private B // private derivation
   {
       // members of D
   }

3. class D: B // private derivation by default
   {
       // members of D
   }
Private members: Accessible within the class only.
The only way to reach them outside the class is through public member functions.
Not accessible through objects.

Protected members: Accessible within the base class + derived class only.
Not accessible through objects.

Private Vs Protected: Absolutely no difference except that protected members are accessible in derived class, remaining behaviour remains same.

Public members: Accessible everywhere in program.
Outside the base class & derived class accessible through object of corresponding class.
The following table provides a summary of how derivation type can modify visibility mode of base class members inside derived class.

<table>
<thead>
<tr>
<th>Base class visibility</th>
<th>Derived class visibility</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Public derivation</td>
</tr>
<tr>
<td>private</td>
<td>Not Inherited</td>
</tr>
<tr>
<td></td>
<td>(inherited base class</td>
</tr>
<tr>
<td></td>
<td>members can access)</td>
</tr>
<tr>
<td>protected</td>
<td>protected</td>
</tr>
<tr>
<td>public</td>
<td>public</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 14.1: Visibility of class members**

(Note: Not inherited does not mean data members are not added to derived class, while calculating the size of derived class be careful of this issue)

Visual depiction of derived class:

![Diagram showing derived class members]

**Figure 14.4: Members of derived class on inheritance**

Explanation: This table shows that virtually everything of base class becomes part of derived class. Access to protected/public members of base class is provided using member functions of derived class. Access to private members of base class thus in turn can be achieved through protected/public member functions of base class.
Inheritance does not work in reverse. Base class cannot access any member of derived class.
A derived class does not inherit constructors & destructors from its base class.

14.6 Constructors in Derived Classes
The constructors play an important role in initializing an object's data members and allocating required resources such as memory. The derived class need not have a constructor as long as the base class has a no-argument constructor. However, if the base class has constructors with arguments (one or more), then it is *mandatory* for the derived class to have a constructor and pass the arguments to the base class constructor. In the application of inheritance, objects of the derived class are usually created instead of the base class. Hence, it makes sense for the derived class to have a constructor and pass arguments to the constructor of the base class. When an object of a derived class is created, the constructor of the base class is executed first and later the constructor of the derived class.

14.7 Destructors in Derived Classes
Unlike constructors, destructors in the class hierarchy (parent and child class) are invoked in the reverse order of the constructor invocation. The destructor of that class whose constructor was executed last, while building object of the derived class, will be executed first whenever the object goes out of scope.
Typical questions

Answer the questions (i) to (iv) based on the following: 4

class PUBLISHER
{
char Pub[12];
double Turnover;
protected:
void Register();
public:
PUBLISHER();
void Enter();
void Display();
};
class BRANCH
{
char CITY[20];
protected:
float Employees;
public:
BRANCH();
void Haveit();
void Giveit();
};
class AUTHOR : private BRANCH , public PUBLISHER
{
int Acode;
char Aname[20];
float Amount;
public:
AUTHOR();
void Start();
void Show();
};

(i) Write the names of data members, which are accessible from objects belonging to class AUTHOR.
(ii) Write the names of all the member functions which are accessible from objects belonging to class BRANCH.
(iii) Write the names of all the members which are accessible from member functions of class AUTHOR.
(iv) How many bytes will be required by an object belonging to class AUTHOR?
Solution:
Question of this nature can be solved very easily by drawing pictorial representation of the situation.

Draw as many tables as there are classes in the question. Each table to have two columns & four rows.

Row1 (name of the class), Row2 (Private members), Row3 (Protected members), Row4 (Public)
Left column (data members), right column (function members).
A computer system stores programs and data in secondary storage in the form of files. Storing programs and data permanently in main memory is not preferred due to the following reasons:

- Main memory is usually too small to permanently store all the needed programs and data.
- Main memory is a volatile storage device, which loses its contents when power is turned off.
Above diagram shows typical stream computation model which uses file stream as mean of communication between program and data file. Input stream supplies data to the program and output stream receives data from the program. 

(Stream can be thought of as a sequence of bytes)

**cin & cout are stream objects** (predefined in iostream.h) used extensively to deal with standard input and output devices (Keyboard/monitor).

**Classes for file handling:**

1. `ifstream` : for handling input files  
2. `ofstream` : for handling output files  
3. `fstream` : for handling such files which require both input & output  

*To use these classes include `<fstream.h>` in your program.*

**I) Text File**: stores information in ASCII. [Each line terminated with EOL (End of Line)].  
**II) Binary File**: stores information as 1 or 0. There is no end of line delimiter in binary files.

*By default files in C++ are treated as text files*

Before starting any operation on a file we need a file pointer to refer the file (Technically called file handler). It is this file handler through which all the file operations takes place.

**How to create a file handler?**

**For reading a file**: `ifstream infile(“test.txt”)`

Explanation – `ifstream` is the class required if sole objective of your program is to read the content of a
file.

Infile : This is name which you have given to your file handler, it could be any valid identifier such as x,y,z or abcd, but infile is the preferred one as it clearly shows the purpose of this file handler.

“test.txt”: Name of the file whose content you want to read.

**For writing in a file :** ofstream outfile(“test.txt”)
Explanation - ofstream is the class required in order to write something to the file.

**For reading & writing operations to be done simultaneously** on a file we need

fstream myfile(“test.txt”)

**What next after creation of file handler ?**

Once we are done with creation of file handler next task is to open the file in appropriate mode. There are several modes in which a file can be opened, but here we will look into IN, OUT, Append & Binary modes only.

There are two ways in which a file can be opened

a) **using constructors of ifstream/ofstream/fstream class.**
   e.g. ifstream infile(“test.txt”)/default mode is IN

   ofstream outfile(“test.txt”)/default mode is OUT

   fstream myfile(“test.txt”,ios::in|ios::binary) // No default mode for this class. specify explicitly whenever want to use this object.

b) **using open method of ifstream/ofstream/fstream class.**
   When opening a file for input/output we can mention the mode in which we want to open it, otherwise default mode is used.

   e.g streamobjeject.open(“filename”,mode)

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ios::in</td>
<td>If FileName is a new file, then it gets created fine as an empty file. If FileName already exists, then it is opened and its content is made available for processing</td>
</tr>
<tr>
<td>ios::out</td>
<td>If FileName is a new file, then it gets created fine as an empty file. Once/Since it gets created empty, you can write data to it. If FileName already exists, then it is opened, its content is destroyed, and the file becomes as new.</td>
</tr>
<tr>
<td>ios::app</td>
<td>If FileName is a new file, data is written to it. If FileName already exists and contains data, then it is opened, the compiler goes to the end of the file and adds the new data to it.</td>
</tr>
<tr>
<td>ios::binary</td>
<td>This causes a file to be opened in binary mode</td>
</tr>
</tbody>
</table>
e.g.
Ifstream infile;
infile.open(“test.txt”);//in mode text file
Ifstream infile;
infile.open(“test.txt”,ios::binary);//in mode binary file

ofstream outfile;
outfile.open(“test.txt”);//out mode

fstream myfile;
myfile.open(“test.txt”,ios::app|ios::binary);append mode binary file

“<<”(write) & “>>”(read) operator :

<table>
<thead>
<tr>
<th>&lt;&lt;</th>
<th>&gt;&gt;</th>
</tr>
</thead>
</table>
| e.g. cout<<“welcome”;
the byte sequence of welcome transferred to screen | cin>>x;
read byte sequence from keyboard and transfer it to x

ofstream outfile(“test.txt”)
outfile<<“welcome”;
the byte sequence of welcome transferred to file. | ifstream infile(“test.txt”)
infile>>x; the byte sequence of welcome transferred to file. |

Testing for errors :

! operator returns non zero value in case of stream errors.

```cpp
ifstream in_file( "test.txt" );
//test for error
if( !in_file )
{   //File wasn\'t opened
cerr << "Cannot open test.txt\n";
    exit( 1 );
}
```

In case of any error in opening or reading the file ! Operator returns non zero value which makes the condition to be true ,error message is displayed and program is terminated.

This logic can be used with any of the ifstream,ofstream,fstream objects to check for error before proceeding with processing of file.
PROGRAM TO READ CONTENT OF A TEXT FILE

// fdisp.cpp: display file contents using ifstream to input from a file
#include <fstream.h>
#include <iomanip.h>
int main()
{
    char ch;
    char filename[25];
    cout << "Enter Name of the File: ";
    cin >> filename;
    // create a file object in read mode
    ifstream ifile(filename);
    if( !ifile )  // file open status
    {
        cerr << "Error opening " << filename << endl;
        return 1;
    }
    ifile >> resetiosflags( ios::skipws ); // do not skip space or new line
    //Comment above line; then execute the program, you will see funny result
    while( ifile )  // while EOF not reached.
    {
        ifile >> ch;  // read a character from file
        cout << ch;   // display character on console
    }
    return 0;

PROGRAM TO WRITE CHARACTERS TO TEXT FILE

// keyin.cpp: Reads all the characters entered and stores the same in the file
#include <fstream.h>
void main()
{
    char ch;
    cout<<"Enter characters..<Ctrl-Z followed by carriage-return to stop>\n";
    ofstream ofile("key.txt"); // opens file in output ASCII mode
    while( cin )  // not end of file
    {
        cin.get(ch);  // read character from console
        ofile << ch;  // write to file
    }
    ofile.close(); // close file
}

Run
Enter characters..<Ctrl-Z followed by carriage-return to stop>
1
A B C .. X Y Z
^Z

Note: The file key.txt has all the above characters except ^Z
**File pointers:**

To facilitate the movement across the file, C++ maintains two file pointers:
- get pointer (also called input pointer)
- put pointer (output pointer)

Get pointer specifies the location where next read operation will be initiated.
Put pointer specifies the location where next write operation will be initiated.

**Default locations:** Depending on the mode (In/Out/Append) in which file is opened, the get & put pointers initial location is decided upon.

```
<table>
<thead>
<tr>
<th>MODE</th>
<th>.get pointer</th>
<th>.put pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Beginning of the file</td>
<td>Not available</td>
</tr>
<tr>
<td>Write</td>
<td>Not available</td>
<td>Beginning of the file</td>
</tr>
<tr>
<td>Append</td>
<td>Beginning of the file</td>
<td>End of the file</td>
</tr>
</tbody>
</table>
```

**File pointers manipulators:** C++ I/O system provides four functions for setting a file pointer to a desired location or for telling the current position of file pointer inside the file.

<table>
<thead>
<tr>
<th>Function</th>
<th>Member of the class</th>
<th>Action Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>fseek()</td>
<td>ifstream</td>
<td>Moves get file pointer to a specific location</td>
</tr>
<tr>
<td>seekp()</td>
<td>ofstream</td>
<td>Moves put file pointer to a specific location</td>
</tr>
<tr>
<td>tellg()</td>
<td>ifstream</td>
<td>Returns the current position of the get pointer</td>
</tr>
<tr>
<td>tellp()</td>
<td>ofstream</td>
<td>Returns the current position of the put pointer</td>
</tr>
</tbody>
</table>
The two `seek` functions have the following prototypes:

```c
istream & seek(long offset, seek_dir origin = ios::beg);
ostream & seek(long offset, seek_dir origin = ios::beg);
```

Both functions set a file pointer to a certain offset relative to the specified origin. The second parameter `origin` represents the reference point from where the offset is measured. It can be specified by using an enumeration declaration (seek_dir) given in the ios class. (See Table 18.3.)

<table>
<thead>
<tr>
<th>origin value</th>
<th>Seeks from...</th>
</tr>
</thead>
<tbody>
<tr>
<td>ios::beg</td>
<td>seek from beginning of file</td>
</tr>
<tr>
<td>ios::cur</td>
<td>seek from current location</td>
</tr>
<tr>
<td>ios::end</td>
<td>seek from end of file</td>
</tr>
</tbody>
</table>

**Table 18.3: File seek origins**

For example, the statement

```c
inFile.seek(20, ios::beg);
```

or

```c
inFile.seek(20);
```

moves the file pointer to the 20th byte in the file. After this, if a read operation is initiated, the reading starts from the 21st item (bytes in file are numbered from zero) within the file. The statement

```c
outFile.seekp(20, ios::beg);
```

or

```c
outFile.seekp(20);
```

moves the file pointer to the 20th byte in the file. After this, if write operation is initiated, the writing starts from the 21st item (bytes in file are numbered from zero) within the file. Consider the following statements:

```c
ofstream outFile("student.out", ios::app);
int size = outFile.tellp();
```

The first statement creates the file stream object `outFile`, and connects it to the disk file, `student.out`. It moves the output pointer to the end of the file. The second statement assigns the value of the put pointer to the integer variable `size`, which in this case represents the number of bytes in the file.

<table>
<thead>
<tr>
<th>Seek call</th>
<th>Action performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>fout.seek(0, ios::beg)</td>
<td>Go to the beginning of the file</td>
</tr>
<tr>
<td>fout.seek(0, ios::cur)</td>
<td>Stay at the current file</td>
</tr>
<tr>
<td>fout.seek(0, ios::end)</td>
<td>Go to the end of the file</td>
</tr>
<tr>
<td>fout.seek(n, ios::beg)</td>
<td>Move to (n+1) byte location in the file</td>
</tr>
<tr>
<td>fout.seek(n, ios::cur)</td>
<td>Move forward by n bytes from current position</td>
</tr>
<tr>
<td>fout.seek(-n, ios::cur)</td>
<td>Move backward by n bytes from current position</td>
</tr>
<tr>
<td>fout.seek(-n, ios::end)</td>
<td>Move backward by n bytes from the end</td>
</tr>
<tr>
<td>fin.seekp(n, ios::beg)</td>
<td>Move write pointer to (n+1) byte location</td>
</tr>
<tr>
<td>fin.seekp(-n, ios::cur)</td>
<td>Move write pointer backward by n bytes</td>
</tr>
</tbody>
</table>

**Table 18.4: Seek calls and their actions**
Sequential Access to a File

A `sequential file` has to be accessed sequentially; to access the particular data in the `file` all the preceding data items have to be read and discarded. A `random file` allows access to the specific data without the need for accessing its preceding data items. However, it can also be accessed sequentially. Organizing a `file` either as sequential or random depends on the type of `media` on which the `file` is organized and stored. For instance, a `file` on a tape must be accessed sequentially, whereas, a `file` on a hard disk or floppy disk can be accessed either sequentially, or randomly. In C++, it is the responsibility of the programmer to devise a mechanism for accessing a `file`.

The `put()` and `get()` Functions

The function `get()` is a member function of the `file` stream class `fstream`, and is used to read a single character from the `file`. The function `put()` is a member function of the output stream class `fstream`, and is used to write a single character to the output `file`. The program `putget.cpp` reads a string from the standard input device, and writes the same to a `file` character by character. A sequential `file` is created and its pointer is positioned at the beginning of the `file`. It is processed sequentially until the end-of-file is encountered.

```cpp
// putget.cpp: writes and reads characters from the file
#include <fstream.h>
void main()
{
    char c, string[75];
    fstream file( "student.txt", ios::in | ios::out );
    cout << "Enter String: ";
    cin.getline(string, 74);
    for( int i = 0; string[i]; i++ )
    {
        file.put( string[i] );
    }
    file.seekg(0); // seek to the beginning
    cout << "Output String: ";
    while( file )
    {
        file.get( c ); // reads a character
        cout << c;
    }
}
```

**Run**
Enter String: Object-Computing with C++
Output String: Object-Computing with C++
write() and read() functions

Unlike put() and get(), the write() and read() functions access data in binary format. In binary format, the data representation in the file and in the system is the same. The difference between the representation of data in text form and binary is shown in Figure 18.9. The number of bytes required to represent an integer in text form is proportional to its magnitude, whereas, in binary form, the size is always fixed irrespective of its magnitude. Thus, the binary form is more accurate, and provides faster access to the file because no conversion is required while performing read or write. The read() and write() functions have the following syntax:

    infile.read( (char *)&variable, sizeof( variable ) );
    outfile.write( (char *)&variable, sizeof( variable ) );

The first parameter is a pointer to a memory location at which the data retrieved from the file is to be stored in case of read() and address at which data is to be written when retrieved from a file in case of write(). The second parameter indicates the number of bytes to be transferred. The program fwr.cpp illustrates the creation and manipulation of binary files.

    // fwr.cpp: use of write and read member of file streams
    #include <fstream.h>
    void main()
    {
        int num1 = 530;
        float num2 = 1050.25;
        // open file in write binary mode, write integer and close
        ofstream out_file( "number.bin", ios::binary );
        out_file.write( (char *)&num1, sizeof(num1) );
        out_file.write( (char *)&num2, sizeof(float) );
        out_file.close();
        // open file in read binary mode, read integer and close
        ifstream in_file( "number.bin", ios::binary );
        in_file.read( (char *)&num1, sizeof(int) );
        in_file.read( (char *)&num2, sizeof(float) );
        cout << num1 << " " << num2 << endl;
        in_file.close();
    }

Run
530 1050.25