

Object Oriented Programming

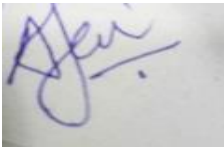
MODULE-5: Templates and Working with Files



Compiled by: Dr. Amita Jain
amita.jain@vsit.edu.in

Certificate

This is to certify that the e-book titled "Object Oriented Programming" comprises all elementary learning tools for a better understating of the relevant concepts. This e-book is comprehensively compiled as per the predefined eight parameters and guidelines.



Signature

Date: 21-11-2019

Dr. Amita Jain

Assistant Professor

Department of IT

⚠ DISCLAIMER: *The information contained in this e-book is compiled and distributed for educational purposes only. This e-book has been designed to help learners understand relevant concepts with a more dynamic interface. The compiler of this e-book and Vidyalankar Institute of Technology give full and due credit to the authors of the contents, developers and all websites from wherever information has been sourced. We acknowledge our gratitude towards the websites YouTube, Wikipedia, and Google search engine. No commercial benefits are being drawn from this project.*

Contents:

1. Templates: Introduction
2. Function Template and examples
3. Class Template and examples
4. Working with Files: Introduction & File Operations
5. Various File Modes
6. File Pointer and their Manipulation

- **Recommended Books:**

1. Object Oriented Analysis and Design by Timothy Budd
2. Mastering C++ by K R Venugopal, Rajkumar Buyya, T Ravishankar
3. C++ for beginners by B. M. Hirwani
4. Effective Modern C++ by Scott Meyers
5. Object Oriented Programming with C++ by E. Balagurusamy
6. Learning Python by Mark Lutz
7. Mastering Object Oriented Python by Steven F. Lott

- **Prerequisites and Linking**

UNIT-V	Pre-requisites	SEM-III	SEM-IV	SEM-V	SEM-VI
Templates & Files	Ip, C++ basics	Data structures and Python	Core Java	Advanced Web Programming, Enterprise Java & Project	Project

I. Templates: Introduction

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

II. Function Templates

A template is one of the features added to C++ recently. It is a new concept which enables us to define generic classes and functions and thus provides support for generic programming. Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.

A template can be used to create a family of classes or functions. For example, a class template for an array class would enable us to create arrays of various data types such as int array and float array. Similarly, we can define a template for a function, say `mul()`, that would help us create various versions of `mul()` for multiplying int and double type values.

In many sorting algorithms, we need to interchange a pair of elements. This simple task is often done by a separate function. For example, the following function swaps integers:

```
void swap(int& m, int& n)
{
    int temp = m;
    m = n;
```

```
    n = temp;
}
```

If, however, we were sorting string objects, then we would need a different function:

```
void swap(string& s1, string& s2)
{
    string temp = s1;
    s1 = s2;
    s2 = temp;
}
```

These two functions do the same thing. Their only difference is the type of objects they swap. We can avoid this redundancy by replacing both functions with a function template.

Like class templates, we can also define function templates that could be used to create a family of functions with different argument types. The general format of a function template in :

```
template<class T>
returntype functionname (arguments of type T)
{
    // .....
    // Body of function
    // with type T
    // wherever appropriate
```

```
// .....  
  
}
```

The function template syntax is similar to that of the class template except that we are defining functions instead of classes. We must use the template parameter T as and when necessary in the function body and in its argument list.

Example

1. The swap Function Template

```
template <class T>  
void swap(T& x, T& y)  
{  
    T temp = x;  
    x = y;  
    y = temp;  
}
```

III. Class Template and examples

A class template works the same way as a function template except that it generates classes instead of functions. The general syntax is

```
template<class T, ...> class X { ... };
```

As with function templates, a class template may have several template parameters. Moreover, some of them can be ordinary non-type parameters:

```
template<class T, int n, class U> class X { ... };
```

Of course, since templates are instantiated at compile time, values passed to non-type parameters must be constants:

```
#include<iostream.h>

template <class T>

class stock

{
    T qty,sp;
public:
    stock(T x,T y)
    {
        qty=x;
        sp=y;
    }

    void showdata()
    {
        cout<<"quantity"<<qty<<endl;
        cout<<"Selling price"<<sp<<endl;
    }

};

void main()
{
    stock <int>s1(15,150);
    stock <float>s2(10.27,20.25);
    cout<<"Information of object 1"<<endl;
```

```
s1.showdata();  
cout<<"Information of object 2"<<endl;  
s2.showdata() ;  
}
```

Class templates are sometimes called parameterized types.

Templates

Source: <https://www.youtube.com/watch?v=BwOQqqRwgpA>

IV. Working with Files: Introduction & File Operations

We have been using the **iostream** standard library, which provides **cin** and **cout** methods for reading from standard input and writing to standard output respectively.

We will see how to read and write from a file. This requires another standard C++ library called **fstream**, which defines three new data types:

Data Type	Description
Ofstream	This data type represents the output file stream and is used to create files and to write information to files.
Ifstream	This data type represents the input file stream and is used to read information from files.
Fstream	This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

To perform file processing in C++, header files `<iostream>` and `<fstream>` must be included in your C++ source file.

File Handling

source: <https://www.youtube.com/watch?v=HcONWqVyvlg>

Opening a File

A file must be opened before you can read from it or write to it. Either the **ofstream** or **fstream** object may be used to open a file for writing and ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of ofstream, ifstream, and ofstream objects.

```
void open(const char *filename, ios::openmode mode);
```

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

V. Various File Modes

Mode Flag	Description
ios::app	Append mode. All output to that file to be appended to the end.
ios::ate	Open a file for output and move the read/write control to the end of the file.
ios::in	Open a file for reading.
ios::out	Open a file for writing.
ios::trunc	If the file already exists, its contents will be truncated before opening the file.

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case it already exists, following will be the syntax:

```
ofstream outfile;  
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows:

```
fstream afile;  
afile.open("file.dat", ios::out | ios::in );
```

Closing a File

When a C++ program terminates it automatically closes flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for `close()` function, which is a member of `fstream`, `ifstream`, and `ofstream` objects.

```
void close();
```

Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (`<<`) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a File

You read information from a file into your program using the stream extraction operator (`>>`) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

Reading File

Source: <https://www.youtube.com/watch?v=lho2EdJgusQ>

Read & Write Example

Following is the C++ program which opens a file in reading and writing mode. After writing information inputted by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen:

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

int main () {

    char data[100];

    // open a file in write mode.
    ofstream outfile;
    outfile.open("afile.dat");

    cout << "Writing to the file" << endl;
    cout << "Enter your name: ";
    cin.getline(data, 100);

    // write inputted data into the file.
    outfile << data << endl;

    cout << "Enter your age: ";
    cin >> data;
    cin.ignore();

    // again write inputted data into the file.
    outfile << data << endl;

    // close the opened file.
    outfile.close();

    // open a file in read mode.
    ifstream infile;
    infile.open("afile.dat");
```

```

cout << "Reading from the file" << endl;
infile >> data;

// write the data at the screen.
cout << data << endl;

// again read the data from the file and display it.
infile >> data;
cout << data << endl;

// close the opened file.
infile.close();

return 0;
}

```

When the above code is compiled and executed, it produces the following sample input and output:

```

$./a.out
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
Zara
9

```

Above examples make use of additional functions from cin object, like `getline()` function to read the line from outside and `ignore()` function to ignore the extra characters left by previous read statement.

File Position Pointers

Both **istream** and **ostream** provide member functions for repositioning the file-position pointer. These member functions are **seekg** ("seek get") for **istream** and **seekp** ("seek put") for **ostream**.

The argument to `seekg` and `seekp` normally is a long integer. A second argument can be specified to indicate the seek direction. The seek

direction can be **ios::beg** (the default) for positioning relative to the beginning of a stream, **ios::cur** for positioning relative to the current position in a stream or **ios::end** for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are:

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );
```

```
// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );
```

```
// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );
```

```
// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

VI. File Pointer and their Manipulations

File Pointers

The C++ input and output system manages two integer values associates with a file.

These are:

- get pointer – specifies the location in a file where the next read operation will occur.
- put pointer – specifies the location in a file where the next write operation will occur.

In other words, these pointers indicate the current positions for read and write operations, respectively. Each time an input or an output operation takes place, the pointers are automatically advances sequentially.

The term pointers should not be confused with normal C++ pointers used as address variables.

Often you may want to start reading an existing file from the beginning and continue sequentially until the end. When writing, you may want to start from the beginning, deleting any existing contents, or appending new records (in which case you can open the file with the `ios::app` mode specifier). These are default actions, so no manipulation of the pointers is necessary.

Sometimes you may have to manipulate file pointers to read from and write to a particular location in a file. The `seekg()` and `tellg()` functions allow you to set and examine the get pointer, and the `seekp()` and `tellp()` functions perform these same actions on the put pointer. In other words, these four functions allow you to access the file in a non-sequential or random mode.

All the `iostream` class objects can be repositioned by using either the `seekg()` or the `seekp()` member function. These functions move the get and put pointers respectively to an absolute address within the file or to a certain number of bytes from a particular position.

The `tellg()` and `tellp()` functions can be used to find out the current position of the get and put file pointers respectively in a file.

The `seekg()` member function takes two arguments:

- Number of bytes to move.
- Reference in the file from which the pointer has to be repositioned.

For example:

```
If stream iFi1;
```

```
iFi1.seekg(10,ios::beg);
```

means, "position the get pointer 10 bytes from the beginning of the file"

The first argument is an integer that specifies the number of bytes positions (also called offset). The second argument is the reference point. There are three reference points defined in the `ios` class:

- `ios::beg` – the beginning of the file.
- `ios::cur` – the current position of the file pointer
- `ios::end` – the end of the file

A negative value can also be specified for the first argument. For example, the following statement moves the file pointer 20 bytes

backward from the end of the file.

```
iFi1.seekg(-20,ios:end);
```

If the seekg() function is used with only one argument, the ios:beg reference point is assumed. For example in the statement:

```
iFi1.seekg(16);
```

ios:beg will be the reference point and hence the get pointer will be positioned 16 bytes from the beginning of the file.

The tellg() member function does not have any arguments. It returns the current byte position of the get pointer relative to the beginning of the file. For example the statement:

```
intiPosition = iFi1.tellg();
```

will result in the variable iPosition having the value of the current position of the get pointer.

The seekp() and tellp() member functions are identical to the above two functions, but they are identified with the put pointer. The seekg() and tellg() member functions are defined in the istream class. The seekp() and tellp() member functions are defined in the ostream class.

Program finds out the number of records in the file billfile.dat by using the seekg() and tellg() functions.

```
#include<fstream.h>
#include<iostream.h>
class bill
{
private:
intiBill_no;
floatfBill_amt;
public:
void getdata()
{
cout<<"Enter Bill number";
cin>>iBill_no;
cout<<"Enter Bill amount";
```



```

cin>>fBill_amt;
}
void showdata()
{
cout<<"Bill number "<<iBill_no<<endl;
cout<<"Bill amount "<<fBill_amt<<endl;
}
};
void main()
{
fstream Fi1("billfile.dat",ios::in);
Fi1.seekg(0,ios::end);
ini iEnd;
iEnd=Fi1.tellg();
cout<<"The size of the file is "<<iEnd<<endl;
cout<<"Size of one record is "<<sizeof(bill)<<endl'
ini iNorec=iEnd/sizeof(bill);
cout<<"There are "<<iNored<<"records in the file"<<endl;
}

```

In the given program, the statement:

```
Fi1.seekg(0,ios::end);
```

is used to reach the end-of-file. The byte position is found by the following statement:

```
iEnd=Fi1.tellg();
```

The value of iEnd is divided by the size of one record to find out the number of records in the file.

Multiple Choice Questions:

1. What is meant by template parameter?

- a) It can be used to pass a type as argument
- b) It can be used to evaluate a type.
- c) It can of no return type
- d) None of the mentioned

2. Which keyword can be used in template?

- a) class

- b) typename
- c) both a & b
- d) function

3. What is the validity of template parameters?

- a) inside that block only
- b) inside the class
- c) whole program
- d) any of the mentioned

4. Which stream class is to only write on files ?

- 1. ofstream
- 2. ifstream
- 3. fstream
- 4. iostream

5. Which stream class is to only read from files ?

- 1. ofstream
- 2. ifstream
- 3. fstream
- 4. iostream

6. Which is correct syntax ?

- 1. myfile.open ("example.bin", ios::out);
- 2. myfile.open ("example.bin", ios::out);
- 3. myfile::open ("example.bin", ios::out);
- 4. myfile.open ("example.bin", ios::out);

Questions

- 1. Explain Templates with Function Templates.
- 2. Explain Templates with class Templates.

3. Design the template class library for sorting ascending to descending and vice-versa.
4. Explain different file operations.
5. Explain various File Modes.
6. Design a class FileDemo open a file in read mode and display the total number of words and lines in the file.
7. Short note on Filepointers and their Manipulations.
8. Design a class to handle multiple files and file operations