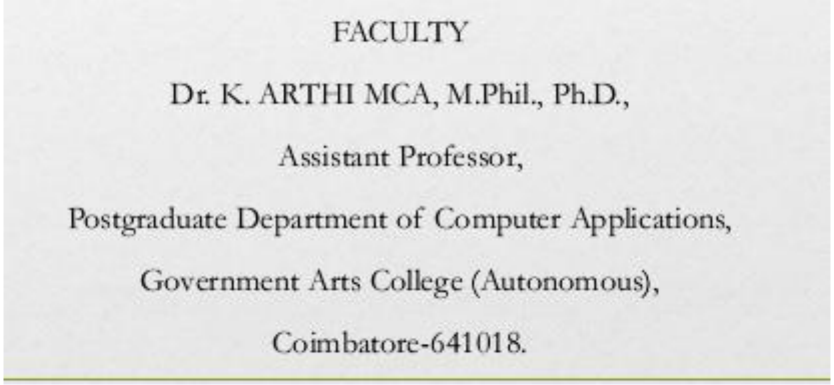
20MCA11C   OBJECT ORIENTED PROGRAMMING AND C++

UNIT V: Streams



FACULTY

Dr. K. ARTHI MCA, M.Phil., Ph.D.,

Assistant Professor,

Postgraduate Department of Computer Applications,

Government Arts College (Autonomous),

Coimbatore-641018.

20MCA11C OBJECT ORIENTED PROGRAMMING AND C++

UNIT I: Principles of Object Oriented Programming: Software Crisis - Software Evolution - Procedure Oriented Programming - Object Oriented Programming Paradigm - Basic concepts and benefits of OOP - Object Oriented Language - Application of OOP - Structure of C++ - Applications of C++ - Tokens, Expressions and Control Structures - Operators in C++ - Manipulators.

UNIT II: Functions in C++: Function Prototyping - Call by reference - Return by reference - Inline functions - Default, const arguments - Function Overloading - Friend and Virtual Functions. **Classes and Objects:** - Member functions - Nesting of member functions - Private member functions - Memory Allocation for Objects - Static Data Members - Static Member functions - Array of Objects - Objects as function arguments - Friendly functions - Returning objects - const member functions - Pointer to members.

UNIT III: Constructors: Parameterized Constructors - Multiple Constructors in a class - Constructors with default arguments - Dynamic initialization of objects - Copy and Dynamic Constructors - Destructors. **Operator Overloading**: Overloading unary and binary operators - Overloading binary operators using friend functions- Overloading the extraction and the insertion operators.

UNIT IV: Inheritance: Defining derived classes - Single Inheritance - Making a private member inheritable - Multiple inheritance - Hierarchical inheritance - Hybrid inheritance - Virtual base classes - Abstract classes - Constructors in derived classes - Member classes - Nesting of classes.

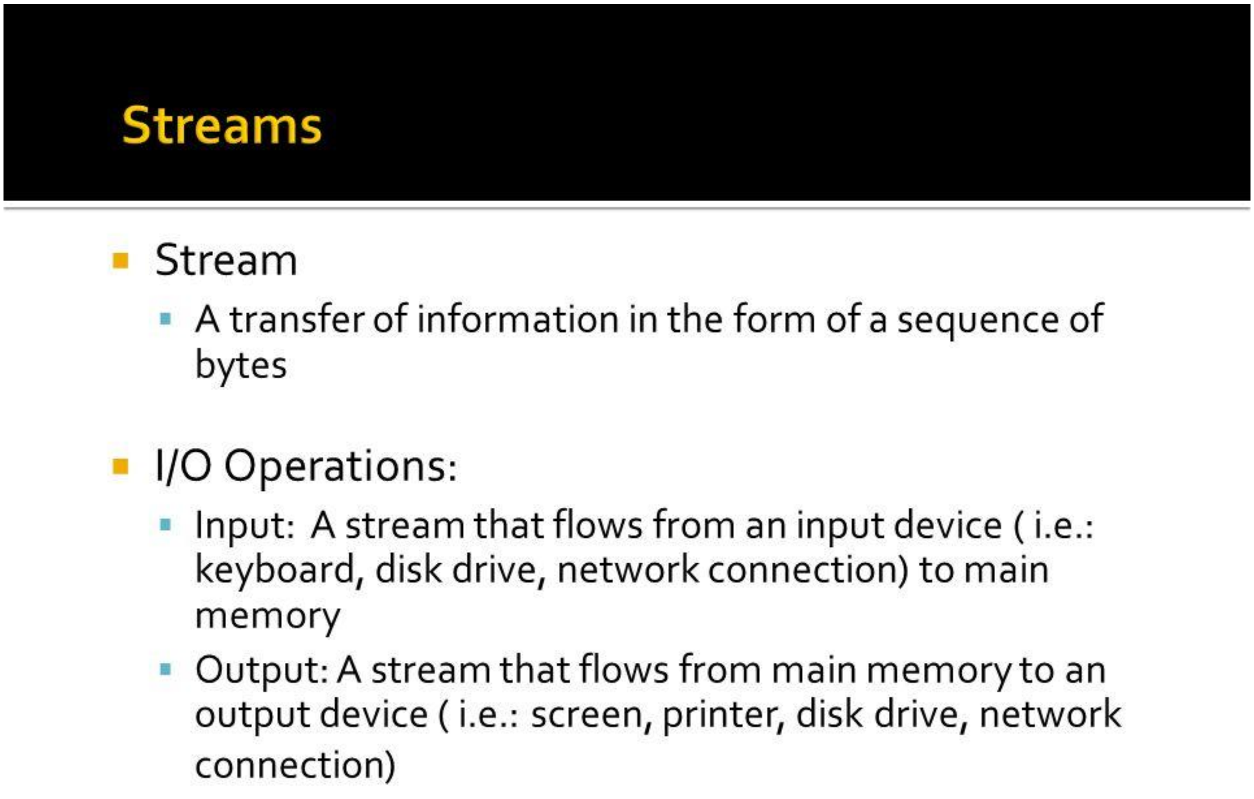
UNIT V: Streams: String I/O - Character I/O - Object I/O - I/O with multiple objects - File pointers - Disk I/O with member functions. Exception handling - Templates - Redirection - Command line arguments.

TEXT BOOKS:

1.E.Balagurusamy, “Object Oriented Programming With C++”, 6th Edition, Galgotia, Publications Pvt. Ltd., 2000.

REFERENCE BOOKS:

1.Herbert Schildt, C++: The Complete Reference, McGraw Hill Inc., 1997.
2.Stanley B. Lippman, Inside the C++ Object Model, Addison Wesley, 1996



Streams

- Stream
  - A transfer of information in the form of a sequence of bytes
- I/O Operations:
  - Input: A stream that flows from an input device (i.e.: keyboard, disk drive, network connection) to main memory
  - Output: A stream that flows from main memory to an output device (i.e.: screen, printer, disk drive, network connection)

Features

- C++ IO is *type safe*. IO operations are defined for each of the type. If IO operations are not defined for a particular type, compiler will generate an error.
- C++ IO operations are based on streams of bytes and are *device independent*. The same set of operations can be applied to different types of IO devices.

C++ provides both the *formatted* and *unformatted* IO functions. In formatted or high-level IO, bytes are grouped and converted to types such as `int`, `double`, string or user-defined types. In unformatted or low-level IO, bytes are treated as raw bytes and unconverted. Formatted IO operations are supported via overloading the stream insertion (<<) and stream extraction (>>) operators, which presents a consistent public IO interface.

To perform input and output, a C++ program:

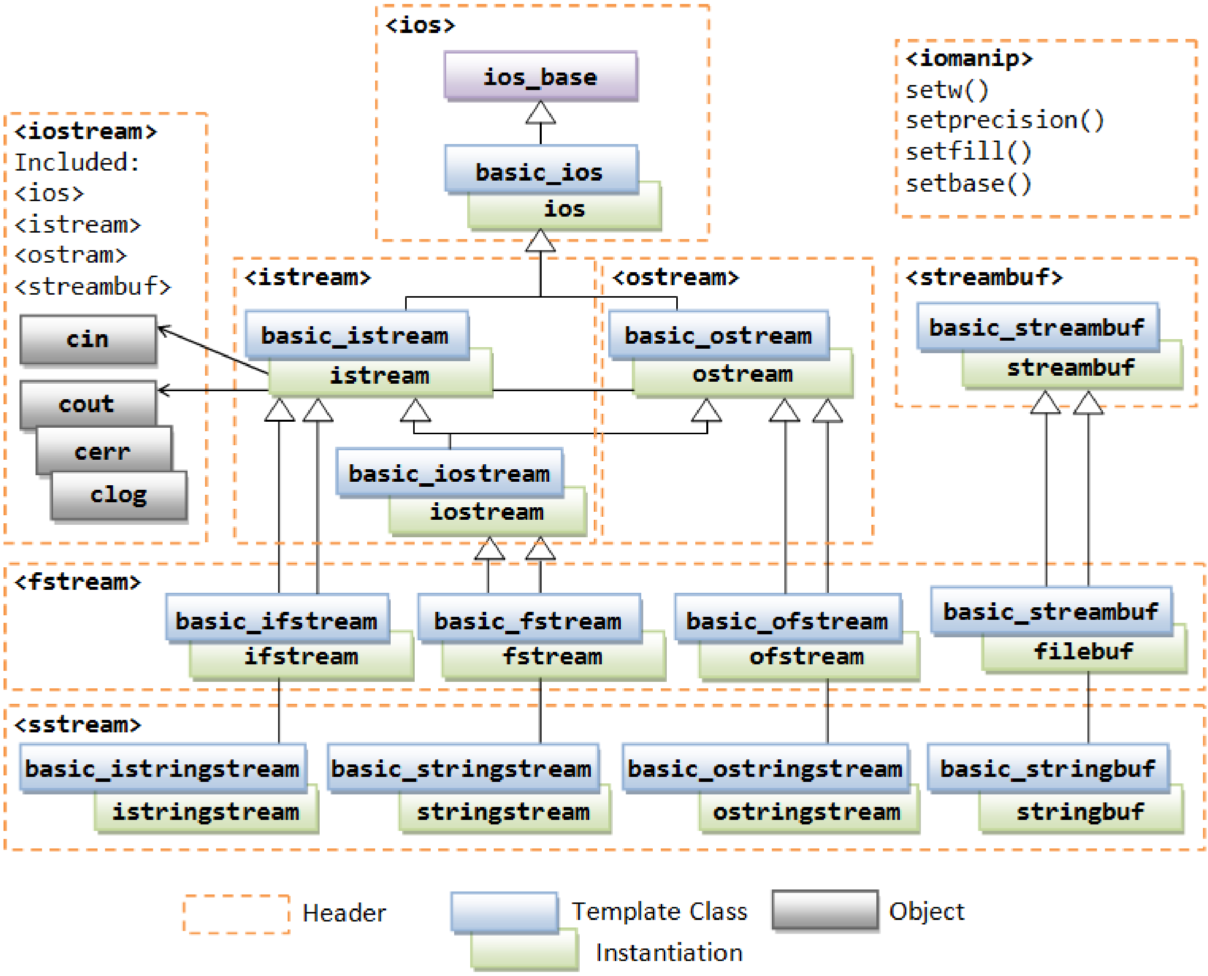
1. Construct a stream object.
2. Connect (Associate) the stream object to an actual IO device (e.g., keyboard, console, file, network, another program).
3. Perform input/output operations on the stream, via the functions defined in the stream's pubic interface in a device independent manner. Some functions convert the data between the external format and internal format (formatted IO); while other does not (unformatted or binary IO).

4. Disconnect (Dissociate) the stream to the actual IO device (e.g., close the file).
5. Free the stream object.

File Input/Output (Header <fstream>)

C++ handles file IO similar to standard IO. In header <fstream>, the class ofstream is a subclass of ostream; ifstream is a subclass of istream; and fstream is a subclass of iostream for bi-directional IO. You need to include both <iostream> and <fstream> headers in your program for file IO.
To write to a file, you construct a ofsteam object connecting to the output file, and use the ostream functions such as stream insertion <<, put() and write(). Similarly, to read from an input file, construct an ifstream object connecting to the input file, and use the istream functions such as stream extraction >>, get(), getline() and read().

File IO requires an additional step to connect the file to the stream (i.e., file open) and disconnect from the stream (i.e., file close).



File Output

The steps are:

1. Construct an `ostream` object.
2. Connect it to a file (i.e., file open) and set the mode of file operation (e.g, truncate, append).
3. Perform output operation via insertion >> operator or `write()`, `put()` functions.
4. Disconnect (close the file which flushes the output buffer) and free the `ostream` object.

File Modes

File modes are defined as static public member in `ios_base` superclass. They can be referenced from `ios_base` or its subclasses - we typically use subclass `ios`. The available file mode flags are:

1. `ios::`**`in`** - open file for input operation
2. `ios::`**`out`** - open file for output operation
3. `ios::`**`app`** - output appends at the end of the file.
4. `ios::`**`trunc`** - truncate the file and discard old contents.
5. `ios::`**`binary`** - for binary (raw byte) IO operation, instead of character-based.
6. `ios::`**`ate`** - position the file pointer "at the end" for input/output.

File Input

The steps are:

1. Construct an `istream` object.
2. Connect it to a file (i.e., file open) and set the mode of file operation.
3. Perform output operation via extraction << operator or `read()`, `get()`, `getline()` functions.
4. Disconnect (close the file) and free the `istream` object.

Exception Handling in C++

One of the advantages of C++ over C is Exception Handling. Exceptions are run-time anomalies or abnormal conditions that a program encounters during its execution. There are two types of exceptions: a)Synchronous, b)Asynchronous(Ex:which are beyond the program's control, Disc failure etc). C++ provides following specialized keywords for this purpose.

try: represents a block of code that can throw an exception.

catch: represents a block of code that is executed when a particular exception is thrown.

throw: Used to throw an exception. Also used to list the exceptions that a function throws, but doesn't handle itself.

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

int main()
```

```
{
   int x = -1;

   // Some code
   cout << "Before try \n";
   try {
      cout << "Inside try \n";
      if (x < 0)
      {
         throw x;
         cout << "After throw (Never executed) \n";
      }
   }
   catch (int x ) {
      cout << "Exception Caught \n";
   }

   cout << "After catch (Will be executed) \n";
   return 0;
}
```

C++ Templates

Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.

The concept of templates can be used in two different ways:

- Function Templates
- Class Templates

Function Templates

A function template works in a similar to a normal function, with one key difference. A single function template can work with different data types at once but, a single normal function can only work with one set of data types.

How to declare a function template?

A function template starts with the keyword **template** followed by template parameter/s inside `< >` which is followed by function declaration.

```
template <class T>
T someFunction(T arg)
{
   ... .. ...
}
```

In the above code, `T` is a template argument that accepts different data types (int, float), and **class** is a keyword.

Class Templates

Like function templates, we can also create class templates for generic class operations.Sometimes, we need a class implementation that is same for all classes, only the data types used are different.

Normally, we need to create a different class for each data type OR create different member variables and functions within a single class.

However, class templates make it easy to reuse the same code for all data types.

How to declare a class template?

```
template <class T>
class className
{
   ... .. ...
public:
   T var;
   T someOperation(T arg);
   ... .. ...
};
```

In the above declaration, `T` is the template argument which is a placeholder for the data type used.
Inside the class body, a member variable `var` and a member function `someOperation()` are both of type `T`.

Command line arguments in C/C++

The most important function of C/C++ is main() function. It is mostly defined with a return type of int and without parameters :

```
int main() { /* ... */ }
```

Command-line arguments are given after the name of the program in command-line shell of Operating Systems.
To pass command line arguments, define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

```
int main(int argc, char *argv[]) { /* ... */ }
```

or

```
int main(int argc, char **argv) { /* ... */ }
```

Properties of Command Line Arguments:

1. They are passed to main() function.
2. They are parameters/arguments supplied to the program when it is invoked.
3. They are used to control program from outside instead of hard coding those values inside the code.
4. argv[argc] is a NULL pointer.
5. argv[0] holds the name of the program.
6. argv[1] points to the first command line argument and argv[n] points last argument.

THANK YOU

This content is taken from the text books and reference books prescribed in the syllabus.