EEET2482/COSC2082

SOFTWARE ENGINEERING DESIGN, ADVANCED PROGRAMMING TECHNIQUES

WEEK 2 – I/O STREAMS & FILE I/O

RMIT UNIVERSITY

Local vs Global variables

- **Local variables**: variables declared
  - ✓ Inside a function
  - ✓ In the list of function parameters
  - ✓ In a code block surrounded by a pair of {}

    They can be accessed only in that function or that block of code (created upon entry destroyed upon exit)

- **Global variables**: variables declared outside all functions
  - ✓ *They can be accessed by all functions (hold values* throughout the program's lifetime)
  - ✓ *Note: Global variables should be used only if necessary (should use local variables if possible).*

Command Line Arguments

- The full format of the main() function is

  ```
  int main(int argc, char const
  *argv[])
  ```

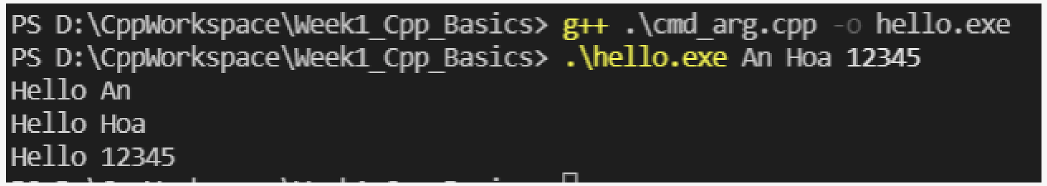
  - **argc**: argument counter, i.e. number of arguments
  - **argv**: array of arguments

    Note: argv[0] is always the name (including path) of the program

- We can pass the arguments (starting from arg[1]) when running a program

  ```
  ./program.exe one two three
  ```

```cpp
/* Save with name cmd_arg.cpp
 Compile: g++ cmd_arg.cpp -o hello.exe
 Run: ./hello.exe name1 name2 123456789
*/

//Get and print out all command-line arguments
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {

    for (int i = 1; i < argc; i++) {
        cout <<"Hello " << argv[i] << "\n";
    }

    return 0;
}
```



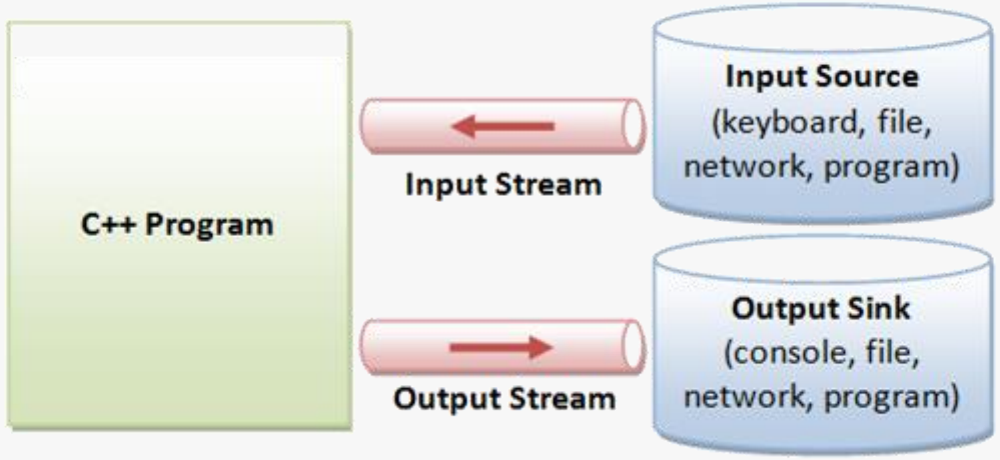
```
PS D:\CppWorkspace\Week1_Cpp_Basics> g++ .\cmd_arg.cpp -o hello.exe
PS D:\CppWorkspace\Week1_Cpp_Basics> .\hello.exe An Hoa 12345
Hello An
Hello Hoa
Hello 12345
```

Good Practice Hints

- Format your code nicely
    - Indent size of 4 spaces, consistent layout of { and }, blank line between two code sections, space between operators, etc.
- File/ function/ constant/ variable names should be meaningful
    - Unless for a temporary purpose e.g. a loop control variable
- Comment your code `// Makes code reading easier`
- Don’t use magic numbers (literals) in code
    - Use named constants for all but simple numbers or literals
    - Use named constants for any literal that may change value in a future version of the code
- Avoid using global variables
- Write functions – each function should have a simple task

C++ Input and Output Streams

- C++ IO are based on *streams*, which are sequence of bytes flowing in and out of the program.
- A stream is linked to a physical device by the C++ I/O system (*provide abstraction to free the programmer from handling the real device at low level IO operations*).


C++ Program
Input Stream
Input Source
(keyboard, file, network, program)
Output Stream
Output Sink
(console, file, network, program)

C++ built-in IO Streams

- When a C++ program begins execution, four built-in streams are automatically opened.

Object	Meaning	Default Device
cin	Standard input	Keyboard
cout	Standard output stream	Screen
cerr	Standard error stream	Screen
clog	Buffered version of cerr	Screen

- **cerr**: un-buffered standard error stream that is used to output the errors *(display the error message immediately).*
- **clog:** buffered version of cerr *(the message is first inserted into a buffer before outputting until the buffer is fully filled or it is explicitly flushed).* It is used to output information to be logged (e.g. for later analysis)

C++ built-in IO Streams

- **cout** and **cerr/clog** are connected to different streams (namely 1 and 2). By default, they are both mapped to the console output, but can be redirected.

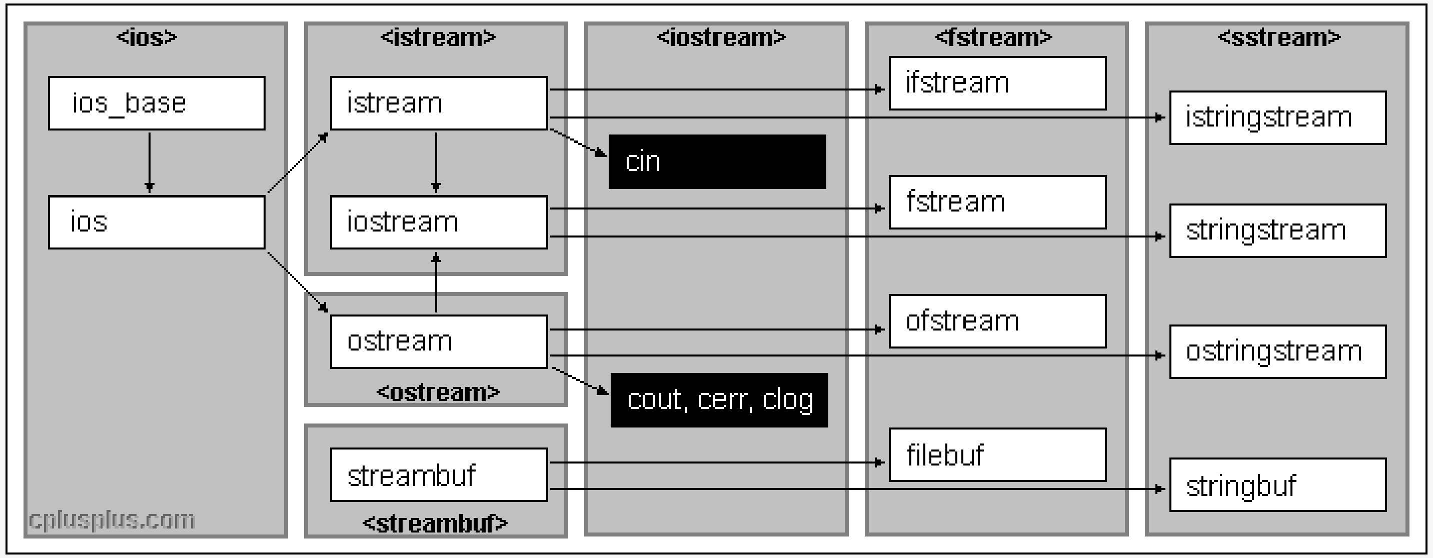
Example:

```cpp
#include <iostream>
int main()  {
    std::cout << "This is cout \n";
    std::clog << "This is clog \n";
    std::cerr << "This is cerr \n";

    return 0;
}
```

```
- Compile: g++ streams.cpp
./a.exe
    Both streams are mapped to the console output (default)

./a.exe > output.txt    or   ./a.exe 1> output.txt
    Map the standard output stream to file output.txt

./a.exe 2> output.txt
    Map the standard error stream to file output.txt
```

Input / Output Library Structure

- The `<iostream>` library is an object-oriented library that provides input and output functionality using streams.



`<ios>`	Base class for streams (type-dependent components).
`<istream>`	Header providing the standard input and combined input/output stream classes.
`<ostream>`	Header providing the standard output stream classes.
`<streambuf>`	Header providing the streambuf buffer class for input/output streams.
`<iostream>`	Declares the objects used for **standard input and output** (eg. `cin` & `cout`).
`<fstream>`	Defines the file stream classes to manipulate **files** using streams.
`<sstream>`	Defines classes to manipulate **`string`** objects as if they were streams.

C++ I/O Stream Formatting

- C++ I/O System allows you to format I/O operations (e.g. number of digits to be displayed after the decimal point).
- There are two ways to do so:

1. Using **IOS member functions**.
2. Using **Stream Manipulators** (special functions)

Formatting using the IOS member functions

The stream has *fields* and *format flags* that control formatting of IO operations.

IOS member functions	Meaning
[.width]()	set the required field **width** (*minimum number of characters to be written*) for displaying *first next* item that is output (effect disappear after that).
[.precision]()	set **precision** for displaying *any* floating point value (maximum number of digits to be written).
[.fill]()	specify the fill character to fill in the blank space (whenever using *width*()).
[.setf](flag [, field])	set a flag for formatting output (returns the previous value of *format flag).*
[.unsetf](flag)	remove the flag setting

Stream Format Flags (ref)

Format state flags	Brief description
ios::skipws	Use to skip whitespace on input.
ios::adjustfield	Controlling the padding, left, right or internal.
ios::left	Use left justification.
ios::right	Use right justification.
ios::internal	Left justify the sign, right justify the magnitude.
ios::basefield	Setting the base of the numbers.
ios::dec	Use base 10 (decimal)
ios::oct	Use base 8 (octal)
ios::hex	Use base 16 (hexadecimal)
ios::showbase	Show base indicator on output.
ios::showpoint	Shows trailing decimal point and zeroes.
ios::uppercase	Use uppercase for hexadecimal and scientific notation values.
ios::showpos	Shows the + sign before positive numbers.
ios::floatfield	To set the floating point to scientific notation or fixed format.
ios::scientific	Use scientific notation.
ios::fixed	Use fixed decimal point for floating-point numbers.
ios::unitbuf	Flush all streams after insertion.
ios::boolalpha	read/write bool elements as alphabetic strings (*true* and *false*)

Example

```cpp
//Precision
std::cout << 3.14159 << " " << 12.3456 << "\n";
std::cout << "precision = 4: \n";
std::cout.precision(4);
std::cout << 3.14159 << " " << 12.3456 << "\n\n";

//Width
std::cout << 10 << " " << 20 << "\n";
std::cout << "width = 5: \n";
std::cout.width(5);
std::cout << 10 << " " << 20 << "\n\n";

//Fill & Width
std::cout << "fill = 'x', width = 5: \n";
std::cout.fill('x');
std::cout.width(5);
std::cout << 10 << " " << 20 << "\n\n";

//Set/Unset Format Flags
std::cout << "setf left justification and showpos, width = 5: \n";
std::cout.setf(std::ios::left, std::ios::adjustfield);
std::cout.setf(std::ios::showpos);
std::cout.width(5);
std::cout << 10 << " " << 20 << "\n\n";

std::cout << "unsetf left justification and showpos, width = 5: \n";
std::cout.unsetf(std::ios::left);
std::cout.unsetf(std::ios::showpos);
std::cout.width(5);
std::cout << 10 << " " << 20 << "\n\n";
```

Get/ Set All Format Flags

- To obtain the current flag settings without making any changes, use:

```
ios::fmtflags f; // declare a variable of type fmtflags
f = cout.flags();
```

- We can also set values for format flags directly with `cout.flags(f)`

See https://whttps://www.cplusplus.com/reference/ios/ios_base/flags/

Example of how to test current flag settings:

```
ios::fmtflags f;
f = cout.flags();           // Get current flag settings

if(f & ios::showpos)
   cout << "showpos is set for cout" << endl;
else
   cout << "showpos is cleared for cout" << endl;

cout.setf(ios::showpos);    // Setting showpos for cout
f = cout.flags();           // Get current flag settings

if(f & ios::showpos)
   cout << "showpos is set for cout" << endl;
else
   cout << "showpos is cleared for cout" << endl;
```

Formatting using the I/O Stream Manipulators

- Alternatively, stream manipulators (special functions) can be used together with insertion(<<) and extraction(>>) operators for IO formatting.
- #include <iomanip> to use following I/O manipulators *(except for endl)*

IOS member functions	Equivalent Stream Manipulators
.width()	**setw**
.precision()	**setprecision**
.fill()	**setfill**
.setf(flag [, field])	**setiosflags**
.unsetf(flag)	**resetiosflags**
NA	**get_time**
NA	**put_time**
NA	**endl** : Inserts a newline '\n' and flushes the buffer

Equivalent Manipulators to set/clear format flags

All stream manipulators below come from std namespace, and can be used *without* including <iomanip> header

Format state flags	Equivalent stream manipulators
ios::skipws	**skipws**
ios::adjustfield	
ios::left	**left**
ios::right	**right**
ios::internal	**internal**
ios::basefield	***setbase*** *(int base)*
ios::dec	**dec**
ios::oct	**oct**
ios::hex	**hex**
ios::showbase	**showbase**: to set, **noshowbase**: to clear
ios::showpoint	**showpoint**: to set, **noshowpoint**: to clear
ios::uppercase	**uppercase**: to set, **nouppercase**: to clear
ios::showpos	**showpos**: to set, **noshowpos**: to clear
ios::floatfield	
ios::scientific	***scientific***
ios::fixed	**fixed**
ios::unitbuf	**unitbuf**
ios::boolalpha	**boolalpha**: to set, **noboolalpha**: to clear

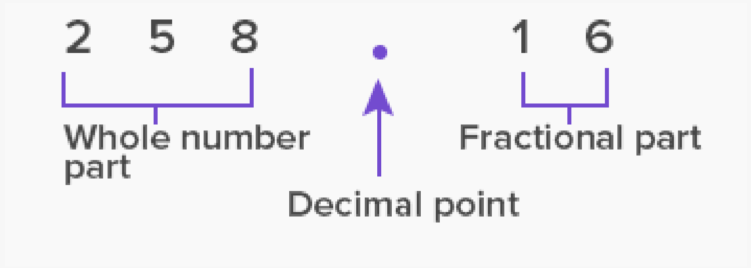
Example

```
double f = 23.14159;
std::cout << std::fixed << std::setprecision(2) << f << "\n";
std::cout << std::setprecision(5) << f << '\n’;
```

Output:

```
23.14
23.14159
```

Note: setprecision(n) applies to the entire number, not the fractional part.
You need to use the fixed-point format to make it apply to the fractional part



C++ File I/O

- C++ can read / write either binary or text files.
  - Text files are human readable (contains the ASCII codes of characters). Binary files are often not human readable and may be compiled executable files, compiled libraries, or data in binary format.
- In C++, files are mainly dealt by using following three classes available in <**fstream**> header file.
  - **ofstream**: allow you to open (or create) a file as an output stream. ie. WRITE
  - **ifstream**: allow you to open file as an input stream ie. READ
  - **fstream**: allow you to open (or create) a file as either an input or output stream. ie. Both READ/WRITE

Some Useful Functions

Function	Description
void **open** (const char* **filename**, ios_base::openmode **mode**)	**Opens** a file with given *mode* and *filename*, associating it with the stream object
void **close**()	**Closes** the file currently associated with the object, disassociating it from the stream (any pending output sequence is written to the file).
bool **is_open**()	Return *true* if a file is open and associated with the *stream* object; *false* otherwise.
istream& **get** (char& c)	**Get character** (extracts a single character from the stream). Returns reference to the stream, or the *end-of-file* value (EOF) if no characters are available in the stream
ostream& **put** (char c);	**Put character** (inserts character *c* into the stream). Returns reference to the stream
istream& **read** (char* s, streamsize n);	**Read block of data** (extracts *n* characters from the stream and stores them in the string array pointed to by *s*).
ostream& **write** (const char* s, streamsize n);	**Write block of data** (inserts the first *n* characters of the string array pointed by *s* into the stream).
istream& **getline** (char* s, streamsize n);	**Get a line**. Extracts up to **n - 1 characters** from a line of the stream.
bool **eof**()	Return *true* if the **End-of-File** has been reached by the last operation; *false* otherwise.

Opening a File

- A file must be opened before you can read from it or write to it
- Standard syntax for open() function:

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

Mode	Description
`ios::app`	All output to the file is appended.
`ios::ate`	Causes a seek to the end of the file i.e. *the current location will be at the end of the file when opened*. I/O operations can however still occur anywhere within the file.
`ios::binary`	Opens file in binary mode. Data sent to the stream is what is exactly written to the file. No text character or substitution takes place at any stage.
`ios::in`	File is capable of input.
`ios::out`	File is capable of output.
`ios::trunc`	Causes any existing file of the same name to be destroyed and the new file to be zero length (overwrite any existing files of the same name)

Writing to and Reading from a File with C++

- **Writing to a File**: we can use stream insertion operator (<<) with **ofstream** or **fstream** object (instead of **cout** object for console output).

- **Reading from a File**: we can use stream extraction operator (>>) with **ifstream** or **fstream** object (instead of **cin** object for console input).

Example: Writing and reading from a file

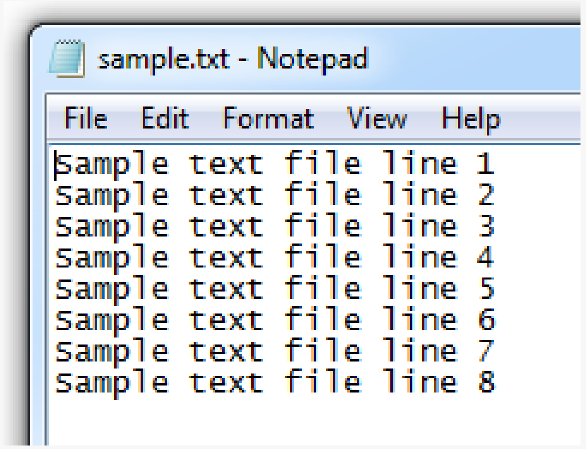
```cpp
int main () {
    char str[100];

    //Create and open a file (use write mode only to create file).
    std::fstream myfile;
    myfile.open("myFile.dat", std::ios::out);

    if (!myfile) {
        std::cerr << "Fail to create/open file \n";
        return -1;
    }

    //Write to file
    int num = 10;
    myfile << num << " Hello World !";
    myfile.close(); // close the file.
    std::cout << "Wrote to the file ! \n" << std::endl;


    //Open for reading and read
    myfile.open("myFile.dat", std::ios::in);
    myfile >> num >> str;

    std::cout << "Read from file: " << std::endl;
    std::cout << num << " " << str << "\n";

    myfile.close(); // close the file.

   return 0;
}
```

Example: Read from console and Write to a file

```cpp
#include <iostream>
#include <fstream>

int main () {
    int your_age;
    char your_name[100];

    std::ofstream myfile;  // declare an ofstream object
    myfile.open("MyFile.txt", std::ios::out); //open file

    if (!myfile) {
        std::cerr << "Fail to open file \n";
    }

    /* Write to file */
    myfile << "Saving to file ..." << std::endl; //flush to write immediately

    /* Read from console and write to file */
    std::cout << "Enter your age: ";
    std::cin >> your_age;
    myfile << your_age << std::endl;

    std::cout << "Enter your name: ";
    std::cin.ignore(1,'\n'); //Ignore previous "\n" character
    std::cin.getline(your_name, sizeof(your_name));
    myfile << your_name << "\n";

    std::cout << "Saved your answers to file !";

    myfile.close();   // close file

    return 0;
}
```

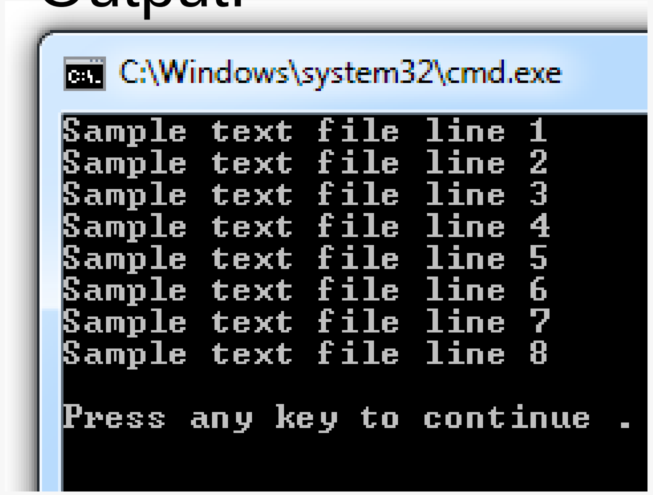
Example: Read each line

Sample text file with 8 line:



```
Sample text file line 1
Sample text file line 2
Sample text file line 3
Sample text file line 4
Sample text file line 5
Sample text file line 6
Sample text file line 7
Sample text file line 8
```

Output:



```
Sample text file line 1
Sample text file line 2
Sample text file line 3
Sample text file line 4
Sample text file line 5
Sample text file line 6
Sample text file line 7
Sample text file line 8

Press any key to continue .
```

```
int main() {
  char buff[256] = {};

  ifstream infile("sample.txt", ios::in | ios::binary);
  if (!infile) {
    cout << "Cannot open file sample.txt" << endl;
    return 0;
  }

  while(!infile.eof()) { // runs until end of file
     infile.getline(buff, sizeof(buff));
     cout << buff << endl;
  }

  infile.close();

  return 0;
}
```