School of Computing and Informatics

BCS362 – Generic Programming with C++

Stream Input/Output and File Processing

1 Stream I/O

1.1 Stream

C++ I/O occurs in **streams**, which are sequence of bytes. For input, the bytes flow from a device (e.g. keyboard, disk drive, network connection) to main memory. For output, bytes flow from main memory to a device (display screen, printer, disk drive or network connection).

1.1.1 iostream library headers

The **iostream** declares basic services required for all stream I/O operations. The **iostream** header defines *cin, cout, cerr* and *clog* objects, which correspond to standard input stream, standard output stream, unbuffered standard error stream and buffered standard error stream.

1.1.2 Stream I/O classes and objects

The *iostream* provides many class templates for performing common I/O operations. We focus on

- **basic_istream** – for stream input operations
- **basic_ostream** – for stream output operations

For each of the class templates *basic_istream, basic_ostream* and *basic_iostream*, the iostream library defines convenient short names

- **istream** – for *basic_istream* that enable char input - this is *cin's* type
- **ostream** – for *basic_ostream* that enable char output - this is the type for *cout, cerr, clog*
- **iostream** – for *basic_iostream* that enable both input and output

1.1.3 Standard stream objects: cin, cout, cerr and clog

Predefined object *cin* is *istream* object that is connected to standard input device (keyboard). The stream extraction operator >> causes a value (of a specific type) to be input from *cin* to memory. The compiler selects appropriate overloaded stream extraction operator, based on the type of the variable. The >> operator is overloaded to input data items of fundamental types, string and pointer values.

The predefined *cout* is *ostream* object that is connected to standard output device (the screen). The stream insertion character << causes the value of the variable to be output from memory to the standard output device.

1.2 Stream output

Formatted and unformatted output capabilities are provided by *iostream*.

- output of standard data types with the stream insertion operator (<<)
- output of character via the **put** member function
- unformatted output via the **write** member function
- output of integers in decimal, octal, and hexadecimal formats
- output of floating point values with various precision, with forced decimal points, in scientific notation (1.2345678e-04) and fixed notation (0.0012345678)

1.2.1 Output of char* variables

```
const char* const name = "UKZN";
cout << "Value is :- " << name;
cout << "\nAddress is :- " << static_cast<const void*>(name) << endl;
```

Listing 1: Output char* (pointer) variables

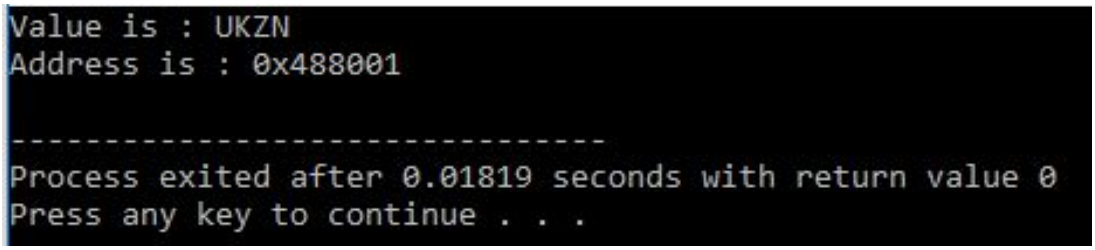

Figure 1: Output of code in Listing 1

1.2.2 Character output using member function put

ostream member function *put* outputs one character at a time. For example

```
cout.put('B');
```

Listing 2: Using **put** to output characters

Call to **put** can be cascaded as

```
cout.put('A').put('B').put('C');
```

Listing 3: Using **put** to output characters

You can still pass the ASCII value to the function **put** and it will display corresponding character as

```
cout.put(65); //This displays A
```

Listing 4: Using **put** to output characters-using ASCII character code

1.3 Stream input

Formatted and unformatted input capabilities are provided by the **istream**. The stream extraction operator (>>) normally skips white space characters (such as tabs, newline and blanks) in the input stream.

1.3.1 Using results of stream extraction as a condition

After each input, the stream extraction operator returns a *reference* to the stream object that received the extraction message. If that reference is used as a condition (e.g. in a while statement continuation condition), the stream overloaded **bool** cast operator function is implicitly invoked to convert the reference into **true** or **false** values, based on the success or failure, respectively, on the last input operation. When an attempt is made to read past the end of a stream, the stream's overloaded **bool** cast operation returns **false**, to indicate end-of-file.

1.3.2 get and getline member functions

The **get** member function without arguments inputs one character from the designated stream (file or keyboard or disk) and return it as the value of the function call. This function returns EOF when end-of-file is encountered. EOF normally has the value -1 and is defined in a header that is indirectly included in your code via stream library like **iostream**.

getline member function reads a line of characters (removes the end-line delimiter -reads the character and discards it). Example

```
#include<iostream>
using namespace std;
int main()
{
  char buffer [100];
  cout << "Enter a sentense" << endl;
  cin.getline(buffer, 100);
  cout << "The sentense entered is \n" << buffer << endl;
}
```

Listing 5: Reading characters using getline

The output is

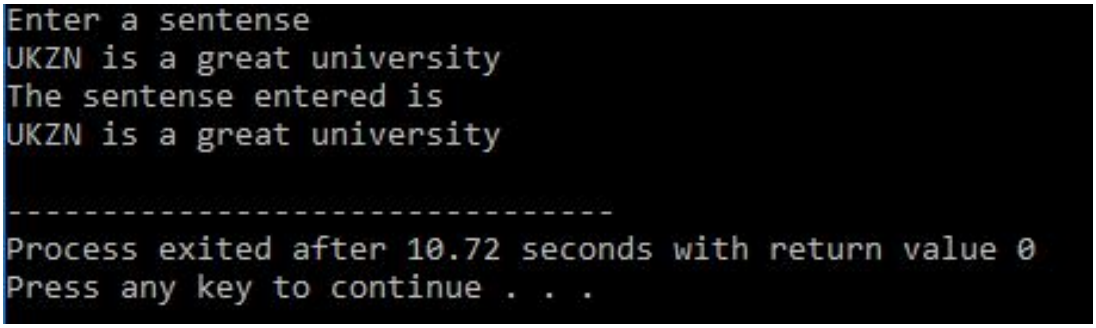

Figure 2: Output of code in Listing 5

2 File processing

Storage of data in memory is temporary. Files are used for data persistence – permanent storage of data. Computers store files in some secondary device such has disk, CD, DVD, flash drives or tapes. In this section we learn how to create C++ programs that create, update and process data files.

2.1 File processing class templates

To perform file processing, C++ headers **iostream** and **fstream** must be included. Header **fstream** includes definitions for the stream class templates

- *basic_ifstream* – a subclass of *basic_istream* for file input
- *basic_ofstream* – a subclass of *basic_ostream* for file output
- *basic_fstream* – a subclass of *basic_iostream* for file input and output

Each has a predefined specialization for *char* I/O. In addition, *fstream* library provides typedef aliases for these template specializations

- **ifstream** is an alias for *basic_ifstream< char >*
- **ofstream** is an alias for *basic_ofstream< char >*
- **fstream** is an alias for *basic_fstream< char >*

2.2 Opening a file

A file can be opened in 6 different modes as

- **ios::app** – Open a file for appending data at the end of the file
- **ios::ate** – Open a file for output and move at-end (ate). Normally used to append data to a file. Data can be written anywhere in the file.
- **ios::in** – Open a file for input (reading)
- **ios::out** – Open a file for output (writing). This is the default mode of file opening
- **ios::trunc** – Discard file content (this is the default mode for ios::out)
- **ios::binary** – Open a file for binary (no text) input and output

To open a file, you create an object of **ifstream** or **ofstream** and pass the file name (full path to file location) and opening mode to the constructor as

```
{
  ofstream output ("data.txt", ios::out);
  for (int i = 0; i < 5; i++)
```

Listing 6: Opening a file

This creates a file named **data.txt** for **output** in the current directory as the source file, and write 0 to 4 to this file. Contents of *data.txt* is

```
0 1 2 3 4
```

You can create an **ofstream** object without opening a specific file. The file can be attached to the object later as

```
    output << i << " ";
  ofstream outfile;
```

Listing 7: Opening a file

After writing to a file, you can close it by calling *close()* function as

```
  outfile.open("data.txt", ios::app);
```

Listing 8: Closing a file

Always close a file immediately it is longer needed by the program.

Assignment: Write a program that will write a student details in a file as

Reg. Number	Name	Programme	Year of Study
12345678	Thando Mkhize	Bsc. Data Science	2
98765432	John Sign Whyte	Bsc. Something	3
45671238	Blessing Nzile	Bsc. Computer	3

Comment line below, code discussed in class.

2.3 Reading data from a sequential file

Files store data so that they may be retrieved for processing when needed. Previous section showed how to write data to a file. This section demonstrates how to read data sequentially from a file.

Creating an *ifstream* objects opens a file for input. The **ifstream** constructor can receive filename and file open mode as arguments as

```
output.close();
ifstream input("data.txt", ios::in);
if(!input) //overloaded ! operator can be used to check if a file was succesfuly opened
```

Listing 9: Opening input file

This line opens a file *data.txt* for input and creates a communication line with the file.

The lines

```
    int x;
    input >> x;
```

Listing 10: Reading from a file

reads the first integer in the file *data.txt* (0) and stores it into the variable x.

To read all the 5 integers from file *data.txt* can be achieved as

```
    input >> x;
    int a, b, c, d, e;
    input >> a >> b >> c >> d >> e;
```

Listing 11: Reading from a file

Given a text file as

```
1 2 3
4 5 6
7 8 9
```

we can use a loop as

```
}
ifstream inputData("reading.txt", ios::in);
int m, n, p;
while(inputData >> m >> n >> p)
```

Listing 12: Reading records from a file

Assignment – Given a file

```
This file contain important data We will read important data only and ignore
the rest. Important data is between [ and ]
but [ and ] are not important
[
1 2 3
4 5 6
7 8 9
]
Important data is between [ and ]
```

Write a program that will read important data from this file and push the data to a vector for further processing.
Comment the line below, this was covered in class.

2.4 Reading and writing quoted data

quoted stream manipulator can be used to read and write quoted data to a file. Consider a file with the content

```
120 "Dr. Raphael Angulu" 27.25
```

we can read the content of this file as

```
ifstream quoted2;
quoted2.open("quoted.txt", ios::in); //remember ifstream objects are opened for input by default
string name;
int number;
float salary;
quoted2 >> number >> quoted(name) >> salary; //quoted works for C++14 or later
cout << "Number : " << number << "\nName : " << name << "\nSalary : " << salary << endl;
```

Listing 13: Reading quoted data from a file

in this code, the first stream extraction operator reads 120 and put it in variable *number*, second reads *Dr. Raphael Angulu* without quotation marks and puts it into *name* and last reads 27.25 and puts it in *salary*.

Similarly, you can write quoted text to output file using the **quoted()** from iomanip library.