INTRODUCTION

In this week we cover the following topics

<u>3.1 STRINGS (C Style)</u>

- We can use arrays to store strings. For example

  ```
  char string[] = “cat”;
  ```

  will create an array of 4 `char`s. Why 4 and not 3? In order for C++ to tell where the end of a string is it appends a special character at the end. This character is `\0` (just like there is `\n`) and is called the *null terminator*.

  If we were to look at the contents of `string` we would see.

  ```
  Array index    0   1   2   3
  Array content  c   a   t  \0
  ```

- Once we have a string within an array we have a lot of power to manipulate it. For example, the following function can be used to determine the length of a string.

  ```
  int stringlength(char string[])
  {
     int length = 0;

     while (string[length] != ‘\0’) length++;

     return length;
  }
  ```

  If we were to call `stringLength` with `string` from before, the function would (and should) return a length of 3 (not 4).

3.1.1 `string.h` Header Library

- This library contains a rich set of functions for manipulating strings. Here is a list of some of its more useful functions. There are many more.
- Return the length of string `s`. See `stringLength` above.

  ```
  int strlen(char *s);
  ```
- Copy `source` string into `dest` string.

  ```
  char* strcpy(char *dest, char *source);
  ```

  E.g.

  ```
  char string1[8], string2[] = “cat”;
  strcpy(string1, string2);
  ```
- Concatenate (append) `source` string to `dest` string.

  ```
  char* strcat(char *dest, char *source);
  ```
- Return a pointer to the first occurrence of character `c` in `s` or the first occurrence of `string` in `s`. Returns `NULL` if nothing is found.

  ```
  char* strchr(char *s, char c);
  char* strstr(char *s, char *string);
  ```
- Compare `string1` to `string2`. If they are the same it returns 0. If `string1` is alphabetically before `string2` it returns a negative number (usually –1) or if `string1` comes alphabetically after `string2` it returns a positive number (usually 1).

```
int strcmp(char *string1, char *string2);
```

E.g.

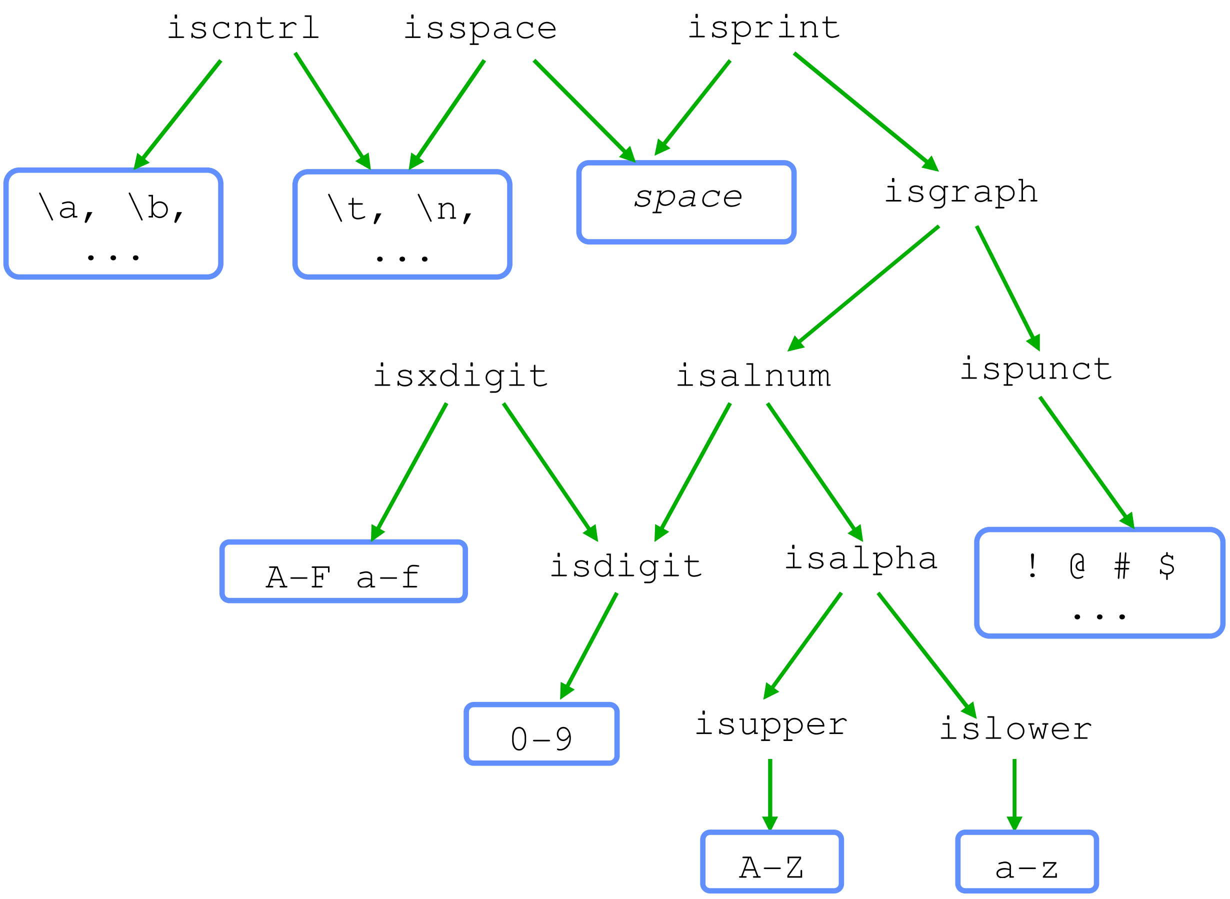
```
if (strcmp(string1, string2) == 0)
   cout << "string1 is equal to string2";
else
   cout << "string1 not equal to string2";
```

3.1.2. - `ctype.h` Header Library

- Most of the functions in this library are predicate functions that test a character and return TRUE if the character has the type being tested for.

- Functions are defined to return `int`'s and take `int`'s, not `char`'s. E.g.
  `int isdigit(int c);`

- The value returned for TRUE is not one, but some non-zero quantity.



- Two other functions:
```
int toupper(int c)   // lower-case
                     // letters to upper
int tolower(int c)   // vice versa
```

  Both functions leave other characters unchanged. E.g.
```
char c = ‘a’;
c = toupper((int) c); /* place ‘A’ in c */
```

- The functions are carefully programmed to be correct, efficient, portable.

Much better to use than writing your own.

3.2 OTHER HEADER LIBRARIES

- Apart from string functions within `string.h` the `stdlib.h` header library contains (at least) three more functions which are of use.

  ```
  int atoi(char *s);
  long atol(char *s);
  double atof(char *s);
  ```

  `atoi` converts a string into an `int`. E.g.

  ```
  char string[] = “127”;
  int number = atoi(string);
  ```

  Similarly, `atol` converts a string to a `long` and `atof` converts a string to a `double`.

- There are many more functions within `stdlib.h` and there are many more libraries in C++. A number of these header libraries are
  - `time.h` which contains many time functions.
  - `math.h` which contains many mathematical functions. Note, many UNIX compilers require the use of the `-lm` switch when compiling with this library.
  - `limits.h` which contains pre-defined constants giving the sizes of various scalar types.
  - `float.h` which contains many pre-defined constants giving the sizes of the two float types.

3.3 COMMAND LINE ARGUMENTS

- Consider the following command typed at the command-line prompt

```
% rm file.cpp
```

- The interesting thing is the use of *command line arguments*, where we feed some information to the program when we start it. In this case the command line parameter is `file.cpp`

- Command line arguments are implemented in C++ using arguments in the `main` function. Up until now we have seen arguments in our own functions but not those for the `main` function. The syntax for them is standardized as follows

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

  `argc` represents the number of parameters received, with the 1st argument (`argv[0]`) being the name of the program. Each argument is stored as an address to a string in `argv`. In other words, `argv` is an array of strings.

- In the previous example, `argv[0]` would contain the string `"rm"` and `argv[1]` would contain the string `"file.c"`.

- The following program illustrates how it works.

```
/****************************************\
   program to print command line arguments
   to screen
\****************************************/

#include <iostream>

using namespace std;

void printArguments(int argc, char *argv[]);

int main(int argc, char *argv[])
{
   cout << "Program name is " <<
        argv[0] << "\nArguments\n";
   printArguments(argc, argv);

   return 0;
}

void printArguments(int argc, char *argv[])
{
   // print arguments to screen

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

If we compiled this program and called it `testarg` then here is an example of what would happen if we ran it.

```
c:\> testarg temp 27.2

Program name is testarg
Arguments
arg 1 = temp
arg 2 = 27.2
```

- We have a problem with `argv` in that it stores things as strings only. Note how the last argument in the previous program is a string of chars, made up of the letters ‘`2`’, ‘`7`’, ‘`.`’ and ‘`2`’. It is not a float variable containing the value 27.2. In many instances though, we want the command line arguments to be number values. This is where the `atoi`, `atol` and `atof` functions become extremely useful.

3.4 CONST

- The `const` keyword can be used to make data read-only. The compiler enforces this and will not compile code if it is trying to write to `const` data.

- The following program illustrates the main uses of `const`.

```
#include <string.h>

const int STRINGSIZE = 20;

const char* getSubString(int index,
                         const char *string);

const char* getSubString(int index,
                         const char *string)
{
   // will NOT compile this line
   string[0] = 'x';

   // will compile this line
   return &string[index];
}
```

```
int main()
{
   char string[STRINGSIZE];
   const char* cptr;

   strcpy(string, "something");
   cptr = getSubString(4, string);

   // will NOT compile this line
   cptr[4] = 'k';

   return 0;
}
```

Compiling this program we get

```
const.cpp: In function `const char*
getSubString(int, const char*)':
const.cpp:6: error: assignment of read-only
location
const.cpp: In function `int main()':
const.cpp:21: error: assignment of read-only
location
```

- If we don't make `cptr` const then we get the following

```
const.cpp: In function `const char*
getSubString(int, const char*)':
const.cpp:6: error: assignment of read-only
location
const.cpp: In function `int main()':
const.cpp:18: error: invalid conversion from
`const char*' to `char*'
```

- Note that we can use `STRINGSIZE` to declare the size of the array because it is a constant, set at compile time. It is no longer a variable.

3.5 STATIC VARIABLES

- C++ has a qualifier to data declarations called static. An example of its use would be
  `static int x;`

- Depending on the location of data being declared it has different effects.
  - *Global variable*. Variables of this type can only be accessed from within the file they are declared. They cannot be accessed from outside the file.
  - *Local function variable*. The variable is automatically initialized to 0. When the function finishes the memory allocated to the variable remains and its contents are unchanged, there to be used the next time the function is called.

3.6 STRINGS (C++ Style)

- C++ has a library called `string` that contains a string object.

- This object is much safer to use than C style array strings as they do a lot of internal error checking, such out of bounds errors.

- It has many member functions and operators supplied with it.

- An example of declaring a `string` data variable called `mystring` would be

```
string mystring;
```

The main operators found with the string are

`=`	copy the contents of a C++ string or C array string to the string.
`[]`	Read and write to an element within a string.
`+=`	Append another C++ string or C array string to the string;
`+`	add one string to another string or array string. Allows for expressions such as `a = b + c;` where `a`, `b` and `c` are strings;

`==`
`!=`
`<`
`<=`
`>`
`>=`
	Boolean comparison operators to determine if two strings are equal, not equal, alphabetically before, etc.

The following member functions are defined as well

`length()`	Length of string
`at(size_t index)`	Return char at index position
`find()`	Find the starting index position of chars, C style strings and C++ style strings within the string.
`compare()`	Similar to strcmp, but between C++ strings, returns 0 for the same, or +/- 1
`c_str()`	Many functions require a C style string as an argument. Call `c_str()` when you want to use a C++ string in these situations.

```
/*********************************************\
   program to print entered strings to screen
\*********************************************/

#include <iostream>

using namespace std;

void getStrings(int *numstringsPtr,
                string* &strings);
void printStrings(int numstrings,
                  const string *strings);

int main()
{
   string *strings = NULL;
   int numstrings = 0;

   getStrings(&numstrings, strings);
   printStrings(numstrings, strings);

   if (numstrings > 0) delete [] strings;

   return 0;
}

void getStrings(int *numstringsPtr,
                string* &strings)
{
   // get strings from user keyboard
   int numstrings;

   cout << "Enter number of strings ";
   cin >> numstrings;

   *numstringsPtr = numstrings;
   strings = new string[numstrings];

   cout << "Enter strings\n";
   for (int i=0; i<numstrings; i++) {
      cin >> strings[i];
   }
}

void printStrings(int numstrings,
                  const string *strings)
{
   // print strings to screen

   cout << "Strings entered\n";
   for (int i=0; i<numstrings; i++) {
      cout << strings[i] << '\n';
   }
}
```

3.7 FILE I/O

- Being able to process files is very important when large amounts of data are to be processed or when the same data is to be reprocessed several times.

- C++ uses an inherited class of the `iostream` called `fstream` that has a very flexible set of file handling functions.

- This is a large and sometimes complex subject. For this course we will restrict ourselves to basic file input and output.

- In order to do file processing there are three concepts you must know
  1. IO Objects
  2. Text vs binary files
  3. Sequential vs direct file access

3.7.1 I/O Objects

- In order to access files C++ uses a concept called *streams*. Streams connect your program to a selected file source.

- C++ has two object types, `istream` (input stream) and `ostream` (output stream). When your program starts it automatically declares the following.
  - `cin` – type `istream` – reads from the standard input, normally from the keyboard (but can be redirected using the UNIX < redirector).
  - `cout` – type `ostream` – outputs to the standard output, normally to the screen (but can be redirected using the UNIX > redirector).
  - `cerr` – type `ostream` – outputs to standard error, normally to the screen, but can be redirected.

- C++ defines two more classess, `ifstream` (which inherits from `istream`) and `ofstream` (which inherits `ostream`), but with extra functions that allow them to read and write to files.

3.7.2 Text Vs Binary Files

- There are two main types of files you need to know about. These are text and binary.

- Printable characters, such as those on the keyboard, take up the first 7 bits of a byte. That is values in the range 0 to 127. Files that are exclusively made up of these are text files. Examples are C++ source files, word and text documents, batch files and so forth.

- Binary files are those made up of bytes using all 8 bits. Examples are executable programs, images and compressed files.

- In this course we will only be concerned with text files.

3.7.3 Sequential Vs Direct File Access

- There are essentially two ways to access data in a file, namely sequential and direct.

- In sequential access you start at the beginning of the file and work your away along until you reach the desired point.

- In direct access you go immediately to the desired point.

- In this course we will only be concerned with sequential access.

3.7.4 Program Example - `filecopy.cpp`

- The following program illustrates file I/O on a text file.

```
#include <string>
#include <iostream>
#include <fstream>

using namespace std;

void getFiles(string &filein,
              string &fileout);
void copyFile(const string &filein,
              const string &fileout);

int main()
{
   string filein, fileout;

   getFiles(filein, fileout);
   copyFile(filein, fileout);

   return 0;
}

void getFiles(string &filein, string
&fileout)
{
   cout << "Please enter file to copy ";
   cin >> filein;
   cout << "Please enter target file ";
   cin >> fileout;
}
```

```
void copyFile(const string &filein,
              const string &fileout)
{
   ifstream fin;
   ofstream fout;
   char ch;

   // Open stream to filein
   fin.open(filein.c_str());
   if (!fin) {
      cerr << "Can't open " << filein
           << " \n";
      exit(1);
   }

   // Open stream to fileout
   fout.open(fileout.c_str());
   if (!fout) {
      cerr << "Can't open " << fileout
           << " \n";
      fin.close();
      exit(1);
   }

   // process file character by character
   while (!fin.eof()) {
      fin.get(ch);
      fout.put(ch);
   }

   // close streams
   fin.close();
   fout.close();
}
```

3.7.5 Opening And Closing File Streams

- To open a stream to a file C++ uses the `open()` function. Its syntax is as follows

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

- `filename` is a string that is the name of the file being processed. It can also be the name enclosed in quotes. E.g.

  ```
  open(“file.txt” , ios::openmode mode);
  ```

- The `openmode` determines how the file will be accessed. There are several modes but the mains ones to be used in this subject are the following

  ```
  ios::app
  ios::in
  ios::out
  ```

  If an `ifstream` stream is opened without the mode included then it will be opened in `ios::in` mode, which is a read mode. Similarly, if an `ofstream` stream is opened without the mode it will be opened in `ios::out` mode, which is a write mode.

  `ios::app` opens an `ofstream` in append mode. E.g
  `fout.open(filename, ios::app);`

- In write and append mode the file is created if it does not exist. In write mode the previous contents of the file are lost and the file is rewritten from the beginning. In append mode, data is appended to the end of the file, preserving the existing data.

- Do not try and read and write to the same file at the same time. Of course you can open a file for writing, close it and then reopen it for reading.

- You should always check to make sure a stream has been opened before processing a file.

- As soon as you are finished processing a file you should close down the stream to it. This is done with the `close()` member function. Its syntax is

  See `filecopy.cpp` for examples of its use.

<u>3.8 I/O CONTROL</u>

3.8.1 Floating Point Format

- When `cout` outputs real numbers to the screen it has 3 possible floating point formats it can use.

scientific	In scientific format the number will be printed with a single digit followed by a decimal point, then the numbers for precision and then an exponent factor of 10 raised to a particular value. Output is made scientific using the `scientific` modifier.
fixed	The number will not be printed in scientific format. Output is made fixed by using the `fixed` modifier.
default	If no format has been chosen then the `iostream` library automatically determines the format to print real numbers.

	• No decimal point: 1.0000 prints as 1 • No trailing zeros: 1.5000 prints as 1.5 • Scientific notation for large/small numbers: 1234567890.0 prints as 1.23457e+09 There is no modifier for getting back to the default format. We can use the `unsetf` member function though to undo the format.

The following code illustrates

```
#include <iostream>
#include <math.h>

using namespace std;

int main()
{
   cout << "Scientific : " << scientific
        << sqrt(999) << "\n";
   cout << "Fixed : " << fixed
        << sqrt(999) << "\n";
   cout.unsetf(ostream::floatfield);
   cout << "Default : "
        << sqrt(999) << "\n";

   return 0;
}
```

```
Scientific : 3.160696e+01
Fixed : 31.606961
Default : 31.607
```

3.8.2 Output Manipulators

- The following output manipulators control the format of the output stream. Some of the manipulators take a parameter and, if they do, they require the inclusion of the `<iomanip>` library.

`setprecision(n)`	This sets the number of decimal points of precision to `n`. For this to work properly need to be in fixed format. The last precision digit printed will be rounded.
`setw(n)`	Sets the minimum width of the output to `n`.
`setfill(ch)`	If the minimum width being outputted is greater than the actual output then the extra characters are padded out with spaces. You can use `setfill` to change the padding character to `ch`
`left` `right`	You `left` or `right` to specify the justification of output.

The following code illustrates

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
   double d = 12.345678;

   cout << fixed;
   cout << setprecision(4) << d << "\n";
   cout << setw(10) << d << "\n";
   cout << setw(10) << setfill('#')
        << d << "\n";
   cout << left << setw(10)
        << setfill('#') << d << "\n";

   return 0;
}
```

```
12.3457
   12.3457
###12.3457
12.3457###
```

3.8.3 I/O Status

- The C++ I/O system maintains status information about each I/O operation. Two states are of particular importance and can be accessed with the following member functions.
  ```
  bool good();
  bool eof();
  ```

- If there has been no trouble with an I/O operation then calling `good()` will return `true`, otherwise `false`.

- The `eof()` function will return `true` if we are at the end of a file, otherwise `false`.

3.8.4 Formatted I/O

- The `>>` and `<<` operators are derived from the `iostream` class and work in exactly the same way with files

- You should always check that the `istream` has correctly read input. However, we have a difficulty with this. When using `cin` a failure to read can be handled by having your code get the user to re-enter the required data. It is difficult, maybe impossible, to do this when reading files. The easiest thing to do is to print an error message and then `exit` the program.

3.8.5 Single Character I/O

- Some quite complex tasks can be done one character at a time.

- Prototypes:
```
istream& get(char &ch);
ostream& put(char ch);
```

3.8.6 Program Example - `filestats.cpp`

- Assume we have the file `data.txt` with the following contents.

```
> cat data.txt
> 5.176389
> 5.21685
> 8.15684
> 6.0916
> 0.1763
> 1.32081
>
```

- It may only contain 6 pieces of data but it can just as easily contain 6,000,000. Let's write a program to determine the mean and standard deviation of the data in this file.

  Remember

$$mean = \frac{1}{n}\sum_{i=1}^{n} x_i$$

  and

$$std = \sqrt{\frac{\sum_{i=1}^{n}(x_i - mean)^2}{n-1}}$$

```
#include <iostream>
```

```
#include <fstream>
#include <string>
#include <math.h>
#include <stdlib.h>

using namespace std;

void getFile(string &file);
void getStats(const string &file, int &size,
              double &mean, double &std);

int main()
{
   string file;
   int size;
   double mean, std;

   getFile(file);
   getStats(file, size, mean, std);

   cout << "Number of elements in file " << file
        << " = " << size << "\n";
   cout << "Mean of file = " << mean << "\n";
   cout << "Standard deviation = " << std << "\n";

   return 0;
}

void getFile(string &file)
{
   cout << "Please enter file to open - ";
   cin >> file;
   cout << "\n";
}
```

```
void getStats(const string &file, int &size,
              double &mean, double &std)
{
   ifstream fin;
   double temp, total;

   // Initialise mean, std and size
   mean = std = 0;
   size = 0;

   // open stream to file
   fin.open(file.c_str());
   if (!fin) {
      cout << "Can't open " << file << "\n";
      exit(1);
   }

   // calculate mean
   total = 0;
   while (!fin.eof()) {
      fin >> temp;
      if (fin.good()) {
         total += temp;
         size++;
      } else if (!fin.eof()) {
         cout << "Unable to read data from "
              << file << "\n";
         exit(1);
      }
   }
   fin.close();

   // must have at least 1 item to work out mean
   if (size == 0) return;
   mean = total / size;

   // must have at least 2 items to work out std
   if (size < 2) return;









   // calculate standard deviation
   fin.open(file.c_str());
   total = 0;

   while (!fin.eof()) {
      fin >> temp;
      if (fin.good()) {
         temp -= mean;
         total += temp * temp;
      }
   }
   fin.close();

   std = sqrt(total / (size-1));
}
```

- Running this program produces the following output

```
Please enter file to open - data.txt

Number of elements in file data.txt = 6
Mean of file = 4.35646
Standard deviation = 3.01845
```

- If we change the data `6.0916` to `6.X` and run it again we get

```
Please enter file to open - data.txt

Unable to read data from data.txt
```