

Introduction to Programming using C++

Lecture Ten: File I/O and Streams

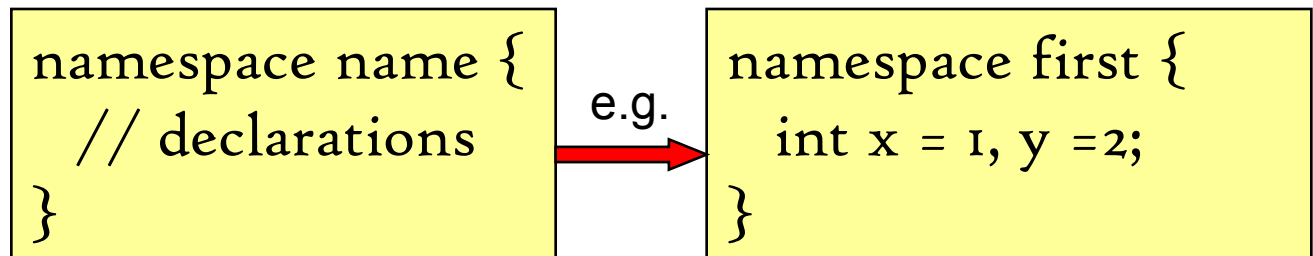
Carl Gwilliam

gwilliam@hep.ph.liv.ac.uk

<http://hep.ph.liv.ac.uk/~gwilliam/cppcourse>

A Note on Namespaces

- Namespaces allow us to localise the names of identifiers (vars, funcs, ...) so as to avoid collisions between them.
 - e.g. if you define a function called `pow()` in your code it could override the `std` library `pow()`.
- To group entities under a name we use the **namespace** keyword:



- A namespace must be declared outside all other scopes.
- All members of the standard C++ library (e.g. `cout`, `endl`, `vector` ...) are within the **std** namespace.
 - We have been using this all along without knowing it!

A Note on Namespaces

- To access elements of a namespace from outside we use the **scope resolution** operator (::) :

```
cout << first::x << endl;
```

- If you have to use elements of a namespace often you can avoid having to scope them each time by using the **using** keyword to:

- bring all elements of the namespace into view:

```
using namespace first;  
cout << x << endl;
```

- bring only particular elements of the namespace into view:

```
using first::x;  
cout << x << endl; // OK  
cout << y << endl; //error
```

- Using multiple namespaces simply brings all set of elements into view (which is fine if they don't clash)!

Streams

- Although you may not realise it, we have already come across streams: **cin** and **cout**!
- These are objects of type **istream** and **ostream** respectively. Together they make up **iostream**.
- There are also complementary **file** streams:
 - **ofstream**: class to write to files
 - **ifstream**: class to read from files
 - **fstream**: combined class to read/write
- Reading and writing from/to files is very similar to using cin/cout; we just need to associate the streams to files!

```
#include<ofstream>
#include<ifstream>
#include<fstream>
```

Opening/Closing a File

- The first thing we want to do with a stream is associate it to a real file i.e. **open** the file. This can be done by:
 - using the **open** member function

```
fstream file; // input or output  
file.open("myfile.txt");
```

- or, more simply, using one of the **constructors**:

```
fstream file("myfile.txt");
```

- Both methods take the filename as a character array.
- It is the same for reading or writing, except you use `ifstream` and `ofstream` respectively.

Opening/Closing a File

- To test if the file has been successfully opened there is an **is_open()** method:

```
if (file.is_open()) {  
    // file is open, do something  
}
```

- Once we're finished with a file we must **close** it:

```
file.close(); // close myfile.txt
```

- Once the file is closed, the stream object **file** can be used to refer to a new file.

File Modes

- Both the methods to open a file mentioned above take an optional **mode** arg, describing how it is to be opened:

```
iostream file(filename, mode);  
file.open(filename, mode);
```

- The various modes, which are in **ios** namespace, are:

ios::in	open for input
ios::out	open for output
ios::app	append to end of current file (output only)
ios::trunc	delete previous content of file (output only)
ios::ate	start at the end of the file (default is start)
ios::binary	open in binary (rather than text) mode

File Modes

- Several modes can be combined using **OR** operator (|):

```
file.open("myfile.txt", ios::out | ios::app);
```

- Often, the mode isn't explicitly specified since there are default modes for the different streams:
 - ofstream: ios::out
 - ifstream: ios::in
 - fstream: ios::in | ios::out
- We will only cover text files in this course, not binary mode files.

Writing to a File

- Writing to a file is done in exactly the same way as writing to the screen with `cout`, using `<<` operator.
- A new line is delimited with either **`endl`** or `\n`.

[myfile.txt]

This is a line
This is another

```
// writing on a text file
#include <fstream>
:
int main () {
    ofstream file("myfile.txt");

    if (file.is_open()) {
        file << "This is a line" << endl;
        file << "This is another.\n";
        file.close();
    } else {
        cout << "Unable to open file\n";
    }
    return 0;
}
```

File States

- There are several useful functions to check the state of the read/write operation:

bad()	True if reading/writing fails (e.g. file not open or no space left on device)
fail()	As bad() but also true if a format error occurs (e.g. read an alphabetical char where num expected)
eof()	Returns true if a file open for reading reaches end
good()	Combination: false if any of above is true

- These are used to see when we should stop reading/writing. E.g.

```
int i = 0;
while (i++ < 10 && ! fiile.bad()) {
    file << "Line of text " << i << "\n"
}
```

File Stream Pointers

- We often need to specify where in a file to **g**et text from or **p**ut text to. We can do this with the functions **seekg()** and **seekp()** respectively.
- These have several forms but the most useful are:
 - setting cursor to beginning of file
 - setting cursor to the end of file
- To see where the cursor currently is we can use the functions **tellp()** and **tellg()**

```
file.seekg(o);  
file.seekp(o, ios::end);
```

Reading from a File

- Reading from a file is done in exactly the same way as reading from the screen with cin, using >> operator.
- Reading stops at any blank character.
- Often used to read tables of input data:

electron	0.000511	-1
muon	0.106	-1
tau	1.7771	-1
proton	0.938	+1
neutron	0.940	0

```
#include <fstream>
:
int main() {           //reading a text file
    ifstream file("mytable.txt");
    string name;
    float mass, charge;

    if (file.is_open()) {
        file.seekg(0); //beginning
        while (!file.eof()) {
            file >> name >> mass >> charge;
            cout << name << "\\t" << mass
                 << "\\t" << charge << endl;
        }
        file.close();
    } else cout << "Unable to open file";
    return 0;
}
```

Reading by Line

- Files can also be read by line using the **getline()** function:

```
getline (ifstream& stream,  
        string&    line,  
        char       delim)
```

- The optional delim arg can be used to specify what is used to delim lines (default is `\n`).

```
#include<fstream>  
:  
int main() {    // read by line  
    ifstream file("myfile.txt");  
    string line;  
  
    if (file.is_open()) {  
        file.seekg(0);  
  
        while (!file.eof()) {  
            getline(file, line);  
            cout << line << endl;  
        }  
        file.close();  
    } else cout << "Unable to open file";  
    return 0;  
}
```

Formatting Output

- We can control the way the output of any stream (be it cout or a file) is formatted in C++.
- This is done via an associated set of format flags, e.g.

In ios namespace

left	left justify output
right	right justify output (default)
dec	output numbers in decimal (default)
oct	output numbers in octal
hex	output number in hexadecimal
fixed	display floats as normal (default)
scientific	display floats in scientific notation
(no)showpos	show leading + sign for +ve numbers
(no)showpoint	show decimal point & trailing 0's for floats

Formatting Output

- There are two separate ways to set these flags.
- Using the **setf()** and **unsetf()** functions of the stream:

```
cout.setf(ios::left);    // left justify  
cout.unsetf(ios::left); // back to right
```

- Again these can be combined using the OR operator (|)
- Using **manipulators** included in I/O stream between << :

```
file << scientific << 10.12345 << “\n”
```

- Note that in this case the **ios** namespace is not needed!
- The formatting applies to all subsequent output statements unless explicitly unset:

```
file << 10.1234 << “\n” //still scientific  
file << fixed << 10.1234 << “\n” //back
```

Precision and Width

- We can also vary the width of the output field and the precision of floats. Again there are two parallel methods.
- By default output only occupies as much space as num of characters, but we can specify a **minminum** width:

```
cout.width(10);           // min width is 10 chars  
cout << setw(10) << 100 << endl; //alternative
```

iomanip
header

- We can change the number of **significant figures** (not d.p.) displayed for floats from the default value of 6:

```
cout.precision(3);           // 3 significant figures  
cout << setprecision(3) << 10.1234 << endl; //alternative
```

10.1

- If use **fixed** & **precision** together we can set num decimal places!
- Whether these alterations apply to all subsequent output or just the next output is implementation dependent ☹.

Formatting Output: Example

```
#include<iomanip>
:
int main() {
    int inum = 100;
    cout << left << setw(15) << "Integer"
        << right<< setw(15) << dec << inum           // decimal
        << setw(15) << oct << inum                     // octal
        << setw(15) << hex << inum                      // hexadecimal
        << setw(15) << showpos << dec << inum << endl;  // with sign
    cout.unsetf(ios::showpos);

    double fnum = 10.1234567;
    cout << left << setw(15) << "Float"
        << right << setw(15) << fnum                   // default (6 s.f.)
        << setw(15) << setprecision(3) << fnum          // 3 s. f.
        << setw(15) << fixed << setprecision(3) << fnum // 3 d. p.
        << setw(15) << scientific << setprecision(3) << fnum << endl; // scientific
    cout.unsetf(ios::scientific); cout.precision(6);
    cout << "Back to normal = " << inum << " " << fnum << endl;

    return 0;
}
```

Integer	100	144	64	+100
Float	10.1235	10.1	10.123	1.012e+01
Back to normal = 100 10.1235				

Stringstream

```
#include<sstream>
```

- Stringstream allows a string to be used as a stream.
- This allows us to extract and insert from/to strings in same way as screen or files, and is useful to convert between nums and strings.
- We insert to stringstream via << and extract from it via >>.
- To retrieve the string from the stringstream we use the **str()** function.

```
#include<sstream>
:
int main() {
    stringstream s1("Carl is ");
    s1 << 27 << " years old";
    cout << s1.str() << endl;

    stringstream s2("28");
    int i; s2 >> i;
    cout << "Next birthday he'll be "
         << i << endl;

    return 0;
}
```

The Debugger (gdb)

- There are debuggers which can help you find errors.
 - On linux with the g++ compiler, the debugger is called **gdb**
- For a **brief** introduction to this we will look at this code, which causes a seg fault:
- For more info you can see:
<http://sourceware.org/gdb/>
- In order to make the most out of the debugger we must compile the code in **debug** mode:

g++ -g debug.cpp -o Debug

```
:  
void func(char* temp){  
    int i = 0;  
    temp[3] = 'F';  
  
    for (i = 0 ; i < 5 ; i++ )  
        cout << temp[i];  
}  
  
int main(){  
    char *temp = "Paras"  
    func(temp);  
    return 1;  
}
```

The Debugger (gdb)

- Some of the more useful gdb commands are:

run <args>	run the program (with any command line arguments)
break <place>	create a place for the program to halt (e.g. line, func,...)
step	execute current line of program and then stop
next	same as step but will execute function call if on line
continue	run until next breakpoint (or end)
backtrace	display chain of function calls (most recent at top)
info locals	display value of any local variables
info args	display value of any arguments to a function
print <var>	print the value of a specific variable
quit	quit gdb (and stop and currently running program)

The Debugger (gdb)

- Finding simple information:

gdb Debug

(gdb) **run**

Starting program: /user1/gwilliam/cppcourse/Examples/lecture10ex/Debug

Program received signal SIGSEGV, Segmentation fault.

0x080486f7 in func (temp=0x80488d8 "Paras") at debug.cpp:9

9 temp[3] = 'F';

(gdb) **backtrace**

#0 0x080486f7 in func (temp=0x80488d8 "Paras") at debug.cpp:9

#1 0x08048758 in main () at debug.cpp:18

(gdb) **quit**

The Debugger (gdb)

- Getting more detailed information:

gdb Debug

(gdb) **break main**

Breakpoint 1 at 0x8048746: file debug.cpp, line 17.

(gdb) **break func**

Breakpoint 2 at 0x80486ea: file debug.cpp, line 8.

(gdb) **run**

Starting program: /user1/gwilliam/cppcourse/Examples/lecture10ex/Debug

Breakpoint 1, main () at debug.cpp:17

```
17      char *temp = "Paras";
```

(gdb) **continue**

Continuing.

Breakpoint 2, func (temp=0x80488d8 "Paras") at debug.cpp:8

```
8      int i = 0;
```

The Debugger (gdb)

(gdb) **step**

```
9      temp[3] = 'F';
```

(gdb) **info args**

```
temp = 0x80488d8 "Paras"
```

(gdb) **info locals**

```
i = 0
```

(gdb) **step**

Program received signal SIGSEGV, Segmentation fault.

0x080486f7 in func (temp=0x80488d8 "Paras") at debug.cpp:9

```
9      temp[3] = 'F';
```

(gdb) **step**

Program terminated with signal SIGSEGV, Segmentation fault.

The program no longer exists.

(gdb) **quit**

That's it!

- Hopefully this course has provided you with an understanding of the basics of C++.
- C++ is a complicated language and I certainly didn't cover everything.
- Please feel free to bug me if you have questions about any of the material in the course.
- Finally, thank you for listening to me prattle on for 10 hours!
- And I leave you with ...

... the best excuse for wasting time:

