Formatted I/O (Input/Output)

PHYS2G03

McMaster University

Formatting I/O

- cout uses stream output. It is designed to make it easy to just dump values to the terminal, e.g.

```
int a=1;
float b=3.141;
std::cout << a << " " << b << "\n";
```

generates:

1 3.141

Stream output

std::cout is an ostream object that outputs to the terminal

std::cerr is an ostream object that outputs error messages to the terminal. Error messages do not get redirected by >

This allows you to have the user see an error message even if they redirect the usual output

cout vs. cerr

testcerr.cpp

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

int main()
   {
   int a=1;
   float b=3.141;

   std::cout << a << " " << b << "\n";
   std::cerr << "This is an error!\n";
}
```

```
> make testcerr
> testcerr
1 3.141
This is an error!
> testcerr > out.txt
This is an error!
> testcerr >& err.txt
>
```

Stream output

You can define your own stream objects to direct output to other places, like files

e.g. ofstream myfile;

myfile.open(“thatfile.txt”);

myfile << a << “ “ << b << “\n”;

You can use any formatting on any stream including output to files.

Format control with streams

streams like cout have extra functionality that let you control what the output looks like in detail using **iomanip**

The syntax is to send the output of special functions to the stream to tell it how to format the next item

i.e.

```
#include <iostream>      // std::cout
#include <iomanip>       // formatting stuff
std::cout << format_function() << thing_to_format;
```

Formatting cout: set width

```
cout << 6
6
```

setw(n) set width of following to n

Note: In some cases this is interpreted as a minimum (if the data doesn't fit)

```
cout << setw(4) << 6 << setw(10) << 2.34;
   6         2.34
1234123456789 0
```

Note: setw() only applies to the next item, after that defaults apply

Formatting cout: set precision

cout << 3.14159

```
3.14159
```

setprecision(n) set precision of following to n

n == total number of digits to display

Default is 6 digits (including before and after .)

cout << setprecision(4) << 3.14156;

```
3.142
```

```
12345
```

Formatting cout: setw vs. setprecision

cout << 3.14159

```
3.14159
```

setprecision(n) set precision of following to n

n == total number of digits to display. It can trump setw and force a larger width to make it all fit

cout << setw(4) << setprecision(5) << 3.14156;

```
3.1416
123456
```

setw(4) interpreted as minimum, 6 used to make sure setprecision(5) can get in the 5 digits

Namespace: simplify using std objects

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



int main()
   {
   std::cout << 3.14159 << "\n";
   std::cout << std::setw(10) << 3.14159 << "\n";
   std::cout << std::setprecision(3) << 3.14159 << "\n";
   std::cout << std::setprecision(10) << 3.14159 << "\n";
}
```

Namespace: simplify using std objects

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

using namespace std;

int main()
   {
   cout << 3.14159 << "\n";
   cout << setw(10) << 3.14159 << "\n";
   cout << setprecision(3) << 3.14159 << "\n";
   cout << setprecision(10) << 3.14159 << "\n";
}
```

Using namespace -- downsides

- Global variables are dangerous and their use is highly discouraged
- Explicit namespaces, e.g. std::cout make it clear where cout comes from
- Using namespace std makes a very large number of words suddenly mean something in C++, e.g. cout,cin,setw
- You can be more restrictive:

  using std::cout;

Namespace: using selectively

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

using std::cout; using std::setw; using std::setprecision;

int main()
   {
   cout << 3.14159 << "\n";
   cout << setw(10) << 3.14159 << "\n";
   cout << setprecision(3) << 3.14159 << "\n";
   cout << setprecision(10) << 3.14159 << "\n";
}
```

Example of formatting

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

using std::cout; using std::setw; using
std::setprecision;


int main()
   {
   cout << 3.14159 << "\n";
   cout << setw(10) << 3.14159 << "\n";
   cout << setprecision(3) << 3.14159 << "\n";
   cout << setprecision(10) << 3.14159 << "\n";
   cout << setw(5) << "frog" << 3.14159 << "\n";
}
```

```
> testformat
3.14159
    3.14159
3.14
3.14159
 frog3.14159
```

Example of formatting

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

using std::cout; using std::setw; using
std::setprecision;

int main()
   {
   cout << 3.14159 << "\n";
   cout << setw(10) << 3.14159 << "\n";
   cout << setprecision(3) << 3.14159 << "\n";
   cout << 3.14159 << "\n";

}
```

What does this one do (and why?)

```
> testformat
3.14159
    3.14159
3.14
```

Example of formatting

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

using std::cout; using std::setw; using
std::setprecision;

int main()
   {
   cout << 3.14159 << "\n";
   cout << setw(10) << 3.14159 << "\n";
   cout << setprecision(3) << 3.14159 << "\n";
   cout << 3.14159 << "\n";

}
```

```
> testformat
3.14159
    3.14159
3.14
3.14
```

What does this one do (and why?)
It reuses the previous precision because there is only one cout object – you are changing the precision of **the** cout object permanently

Formatting cout: combinations

cout << 3.14159

```
3.14159
```

setw(n) and setprecision(m)

cout << setw(10) << setprecision(4) << 3.14156;

```
      3.142
```

```
1234567890
```

setw specifiers are “used up” on the next item printed. setprecision stays.

Defaults

You can also call functions to set the precision etc…
cout.precision(n);

```
std::cout.precision(3);
cout << 3.14159 << “\n";
cout << 12.345678 << “\n";
```

```
3.14
12.3
```

Exponential Notation

For large numbers exponential notation is useful

The default is to switch to this for very small or very large numbers

You can specify to always used fixed (non exponential) or scientific notation

In fixed or scientific, precision sets the number of digits after the decimal place (not the total digits)

Default, Fixed, Scientific

```
double a = 3.1415926534;
double b = 2006.0;
double c = 1.0e-10;

std::cout.precision(5);

std::cout << "default:\n";
std::cout << a << '\n' << b << '\n' << c << '\n';

std::cout << "fixed:\n" << std::fixed;
std::cout << a << '\n' << b << '\n' << c << '\n';

std::cout << "scientific:\n" << std::scientific;
std::cout << a << '\n' << b << '\n' << c << '\n';
```

```
default:
3.1416
2006
1e-010
fixed:
3.14159
2006.00000
0.00000
scientific:
3.14159e+000
2.00600e+003
1.00000e-010
```

Aside

std::cout is an object

- There are 3 of them: cout, cin and cerr
- They are global object variables:

  std::ostream cout, cerr; std::istream cin;
- There is only one copy of each and they have a state – they remember what the precision and other options you set.
- This can be confusing because printing out text in one part of your program could affect any later prints in principle, even in different function
- For comparison: std::ifstream myfile1, somefile;
- You can make as many of these objects as you like and they remain independent from each other. If you set precision etc… for myfile1, other files don't care.

Old school C: printf

C and C++ support the original C print function:
printf defined in stdio.h, e.g.

```
#include <stdio.h>

int main() {
    printf("Hello world!\n");
}
```

printf is easier for formatting

printf function usage:

printf(“*formatstring*”, var1, var2, var3, …);

printf can have as one or more arguments

The first argument is a string with the formatting and all the rest are variables or expressions to be printed. A major bonus is that any formats you set are not remembered – no state.

Printf format

For every variable or expression you include a special character indicating how to print it based on its type:

e.g.

%d or %i	decimal integer
%f	float (real number)
%e	scientific notation
%c	character
%s	string
%p	pointer

printf formatting

You can set the width and precision immediately in the format string with numbers between the % and the type

%nd integer n characters wide

%n.pf float n width, p digits precision

e.g. printf(“%4.2f”, 3.14159);

`3.14`

%10s string 10 characters wide

Formatting width

Add an integer after the %

```
printf("%9d", 60);
        60
123456789
printf("%10f", 500.123456789);
500.123457
1234567890
```

Numbers to show precise width of text
Not printed!

Decimal places (precision)

Add a period then an integer after the % to set places

```
printf("%.3f",3.1415926535897);
3.142
```

```
  123
```

Width and precision

Add an integer for width then a period then an integer for decimal places after the %

```
printf("%10.4f", 3.1415926535897);
    3.1416
1234561234      10 total width, 4 decimal places
```

Advanced formatting

%ld long integer

%0nd integer, width n with 0's not spaces

%x hexadecimal %o octal

%+e scientific with sign

%g most compact of %f or %e to write

%*d use first value to specify width

%% write a percent sign %

http://www.cplusplus.com/reference/cstdio/printf

printf for floats with %g

Code

```
for (i=0;i<10;i++) {
    printf("%g\n",pow(10,i));
}
```

Output

```
1
10
100
1000
10000
100000
1e+06
1e+07
1e+08
1e+09
```

valid C and C++ program

```
#include <stdio.h>
int main()
{
  printf ("Characters: %c %c \n", 'a', 65);
  printf ("Decimals: %d %ld\n", 1977, 650000L);
  printf ("Preceding with blanks: %10d \n", 1977);
  printf ("Preceding with zeros: %010d \n", 1977);
  printf ("Some different radices: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);
  printf ("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);
  printf ("Width trick: %*d \n", 5, 10);
  printf ("%s \n", "Some text");
}
```

```
Characters: a A
Decimals: 1977 650000
Preceding with blanks:       1977
Preceding with zeros: 0000001977
Some different radices: 100 64 144 0x64 0144
floats: 3.14 +3e+000 3.141600E+000
Width trick:    10
Some text
```

printf vs. cout

printf is a lot more compact if you want to format your output in detail. It also has no state so you don't accidentally change precision etc…

Many people use printf for formatted output regardless of whether they use C or C++

Output: fprintf, sprintf

There are many variants on printf

fprintf prints to a stream/file:

- int fprintf(FILE * stream, char * format, ...);

stream can be predefined, e.g.

stdio	the terminal – just like printf
stderr	an error message to the terminal
a file	must be opened first (more later)

sprintf prints to a character array

- int sprintf(char * str, char * format, ...);

str	a character array to be written to

Making text strings safely

Character arrays have a finite size. To be safe, you should let the function know the size – use **snprintf**

int snprintf(char * str, int size, char * format, ...);

```
char text[100];
snprintf(text,sizeof(text),"Item %d is a %s",5,"box");
printf("%s\n",text);
```

Prints the text:

```
Item 5 is a box
12345678901234 5
```

Making text strings safely

Arrays of characters (called string) can be the target for formatted output:

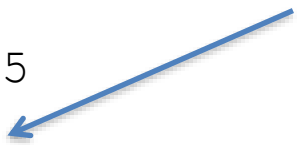
Note that a \0 is automatically appended to the mark the end of the string.

This means that only size-1 characters can be safely stored before the \0

```
char text[15];
snprintf(text,sizeof(text),"Item %d is a %s",5,"box");
printf("Size of text: %i\n",sizeof(text));
printf("%s\n",text);
```

Prints the text:

```
Size of text: 15
Item 5 is a bo
123456789012345
```



A zero character
'\0' is stored here

Making filenames

Strings without spaces are ideal for filenames. Running simulations you can have many outputs that need distinct filenames

```
char filename[200];
int iOut=12;
snprintf(filename,sizeof(filename),
   "Output.%.5i",iOut);
printf("%s\n",filename);
```

Prints the filename:

```
Output.00012
123456712345
```

Example application: output files

```
void writeoutput( int istep, char *bigdata, int ndata) {
    char filename[256];
    sprintf( filename,"output.%05i",istep);
    ofstream outputfile;
    outputfile.open(filename);
    outputfile.write( bigdata,  ndata);
    outputfile.close();
}
```

Example application: output files

```
void writeoutput( int istep, char *bigdata, int ndata) {
    char filename[256];
    sprintf( filename,"output.%05i",istep);
    ofstream outputfile;
    outputfile.open(filenar
    outputfile.write( bigda
    outputfile.close();
}
```

i.e. if istep = 10
This function makes a filename:
output.00010
And then writes ndata bytes of data from the location bigdata to it and returns

Reminder: C++ strings

- To address the safe size aspect of C strings that use character arrays, C++ came up with std::string
- std::string a; a is an object, it knows how big it is a.size() and gets bigger when required without user intervention
- To get an ordinary char array back use:

  a.c_str()

streams and std::strings

```
#include <string>       // std::string
#include <iostream>     // std::cout
#include <sstream>      // std::stringstream
int main () {
  std::stringstream ss;
  ss <<  "Example string " << 42 << 1e-3;
  std::string s = ss.str();
  std::cout << s << "\n";
  return 0;
```

Make a std::string using stringstream

Example string 42 0.001

Stream input in C++: cin

- cin (streamed input) usually does a good job interpreting user input
- It translates everything up to the next whitespace (space, tab or newline) into the type you extract from the stream

e.g.

```
int a;
std::cin >> a;
```

Whatever you type is interpreted an integer

cin pitfalls

cin is very literal. int doesn't want to see decimal places or letters, e.g.

int a; float b;

std::cin >> a >> b;

Input:

```
3.76 5
```

Result:

a=3 stop at . because cannot use . for an int

b=0.76

5 is wasted

cin errors

cin is very literal. int doesn't want to see decimal places or letters, e.g.

int a; float b;

std::cin >> a >> b;

Input:

`hello` hello cannot be interpreted as int

Result:

a=junk

b=junk

cin is in a failed state cin.fail() == 1

cin can't be used now!

cin failed state

If you want to be sure the input was ok you need to check

```
#include <iostream>
int main() {
    int a,b;
    std::cout << “Input two integers\n”;
    std::cin >> a >> b;
    if (std::cin.fail()) {
        std::cout << “Input is bad.  Panic!”;
        return -1;
    }
```

cin failed state clear

If you want to try again after bad input you need clear cin and delete the bad input (this is ugly)

```
#include <iostream>
int main() {
	int a,b;
	std::cout << “Input two integers\n”;
	for (;;) {
	  std::cin >> a >> b;
	  if (std::cin.fail()) {
	      std::cin.clear();
	      std::cin.ignore(INT_MAX,'\n');
	      std::cout << “Input was bad try again.”;
	   }
	}
```

Old school C / C++: scanf

C provided the scanf function

It is just like printf but reads values instead

To read values it needs pointer, not just values, so it can store the values entered

e.g.

```
int a,b;
scanf("%d %d\n", &a, &b);
```

scanf pitfalls

scanf is clunkier than cin and has similar pitfalls

scanf returns how many values were successfully read, e.g.

nread = scanf(“%d %d\n”,&a,&b);

if (nread < 2) printf(“Read failed!\n”);

if scanf fails, stdin does not become unusable, however you still need to remove the unwanted input by hand

Input: fscanf, sscanf

There are variants on scanf

fscanf reads from a stream:

- int fscanf(FILE * stream, char * format, ...);

stream can be predefined, e.g.

stdin the terminal – just like scanf

a file must be opened first (more later)

sscanf read from a string (character array)

- int sscanf(char * str, char * format, ...);

str a character array to be read from

Parsing (cin, scanf)

- Interpreting text is also called parsing
- It is central to computer science and programming languages. A key part of all compilers is the code that parses your .cpp source file and tries to make sense of it.
- The command line (shell) also needs to parse to interpret command you type.
- Natural language parsing is major research area (Turing test)