11. Further Input/Output

We introduced the use of the `iostream` library earlier. Now we will cover the contents of this library in a little more detail.

11.1 Default stream input

The inclusion of the interface file `iostream` provides a program with access to the library of routines to communicate with the world beyond the program. It connects a default stream called `cin` to the keyboard, enabling the program to access characters being typed there.

It also provides the operator >> which is defined thus:

```
cin >> variable
```

takes a stream of characters from the keyboard, and converts them to the internal representation of a value of the type of the variable. This operator is defined for integers, reals and characters (including strings). The **value** of the expression is `cin`. Thus, when more than one operator appears such as

```
cin >> variable1 >> variable2
```

the total expression is calculated from left to right (the first part `cin >> variable1` becoming `cin` so that the second operation is `cin >> variable2`).

But what happens if there is a problem with input?

Recall that a stream is a continuous sequence of characters. If we ask for an integer to be input and we type in

```
41 Brown St Wollongong 2500<ret>
```

then `cin` would start getting characters one at a time from the stream, decoding them into a **valid** integer value. It would stop at the space, passing the value 41 to the variable specified. The next time `cin` was used decoding would begin at the space. The space would be ignored (skipped) until a non-whitespace character is encountered. But suppose the next item we want to read is another integer. `cin` would encounter the `'B'` and immediately return with a failure to decode an integer. This means that if a `cin` expression involves more than one variable and the first read fails, then the subsequent reads are not even performed.

There are several functions provided in the `iostream` library to interrogate the state of a stream. For `cin` they include

```
int cin.good();
```

The prefix `cin.` indicates the function belongs to the stream `cin`. This syntax is a part of the **class** concept in C++ which will not be covered here. We will just consider this to be an extension of the naming of functions.

If the function `good()` is true, the last input attempted was successful, and further use of the stream is possible. If the function `good()` is false, the last input was unsuccessful, due to encountering a character not compatible with the type being read. For example, trying to read a second integer in the case above. Once input is not good, you cannot read any more items from the stream. Doing so will not advance through the stream, nor change variable values. You can clear the error using

```
void cin.clear();
```

(This is not exactly the function, but this description will suffice for now.)

Once the error flag is cleared, the rest of the input line can be read in, one character at a time, until the newline is encountered. Then further input can be received.

The function

```
int cin.eof();
```

indicates that the (logical) end of the stream has been encountered – and the stream is no longer available. This is often used when reading from streams attached to data files (see later this section), but can also be used from `cin`. The problem is that different computers use different ways of indicating end-of-file on the keyboard. Linux will consider control-d (control key and d pressed together) as end-of-file. But MS-DOS wants control-z. It is also advisable to enter this end-of-file character at the beginning of a line of input for consistent results.

Even with the variations, using end-of-file is useful. Suppose we want to average a set of floats but we don't know how many values there are going to be.

```
cin >> x;
no_val = 0;
while (!cin.eof())
{
     sum += x;
     no_val++;
     cin >> x;
}
average = sum/no_val;
```

Note that, once the end-of-file is encountered, no further input from `cin` is possible. Do **not** attempt to clear this flag.

If `cin` fails for reasons other than end-of-file, the program will hang (or go into an endless loop, not advancing the position in the stream). If instead we use

```
cin >> x;
no_val = 0;
while (cin.good())
{
     sum += x;
     no_val++;
     cin >> x;
}
average = sum/no_val;
```

then both end-of-file and fail will terminate the input.

11.2 Output manipulation including formatting

Sometimes the simple output provided by cout is not enough. For example, you may want to output some real numbers which represent dollars and cents. This means that two decimal places should be shown not 1 or 0 or 3. Or you might want to align the output of a program into a tabular form where several lines of output should have values in neat columns.

There exists a set of output manipulators and flags for controlling the format of output. There are two ways of using these manipulators:
(i) by direct calls to the functions which are included in the interface `iostream`;
(ii) by inserting references to them into the stream itself and including the header `iomanip`.

Here are the available manipulators using method (ii).

`setw(`*`w`*`)`	display the next value in a field of width *w*(default is 1).
`setprecision(`*`p`*`)`	display values with precision *p*(number of digits after the decimal point) default is 6.
`setiosflags(`*`flist`*`)`	set the formatting flags in *`flist`* which is a sequence of one or more flags, separated by | (the vertical bar) as in `flag`$_1$ | `flag`$_2$ | `...` | `flag`$_n$

Here are some of the flags:

`ios::showpoint`	always display the decimal point in real output
`ios::fixed`	use fixed space form
`ios::scientific`	display in floating-point form
`ios::left`	left-justify within the field
`ios::right`	right-justify within the field

For example,

```
a = 8./3.;
b = 9./3.;
cout << "(" << a << ")\n"
          << "(" << b << ")\n";
```

yields output

```
(2.666667)
(3)
```

while

```
cout << setiosflags(ios::showpoint | ios::fixed)
          << "(" << a << ")\n"
          << "(" << b << ")\n";
```

would yield
```
     (2.666667)
     (3.000000)
```

and
```
     cout << setiosflags(ios::showpoint | ios::fixed)
                << setprecision(3)
                << "(" << a << ")\n"
                << "(" << b << ")\n";
```

would produce
```
     (2.667)
     (3.000)
```

Using `setw` yields this sort of manipulation.

```
     cout << setiosflags(ios::showpoint | ios::fixed)
                << setprecision(3)
                << "(" << setw(10) << a << ")\n"
                << "(" << b << ")\n";
```
giving

```
     (2.667)
     (3.000)
```

Note that `setw()` only affects the next output while the others continue in effect until changed. Note also right justification is default. If a value does not fit into the field width prescribed, it will extend beyond the width specified.

The flags can be set (and unset) using the functions `setf()` and `unsetf()` prefixed by `cout`. Thus

```
     cout.unset(ios::fixed)
```

would change the format to the default which is a mixture of scientific and fixed depending on the magnitude of the exponent.

Precision and width can be set using the functions

```
     int cout.precision(p);
     int cout.width(w);
```

where the return values are the previous settings (so that they can be saved and restored).

11.3 Standard error

Although it is not significant on personal computers, a practice on mainframe computers is to direct error messages from programs to a separate output stream called **`cerr`**, or standard error. On UNIX systems, for example, where redirection of output to a file is possible, standard error can still appear on your screen while other output is sent to a disk file.

Standard error is just another stream, so it can also be manipulated using the `iomanip` functions and manipulators without affecting `cout`, and vice-versa.

It is a good habit to get into.

11.4 Other streams

We mentioned earlier that the streams `cin`, `cout` and `cerr` were **default** streams automatically linked to the keyboard and the screen. The programmer can, however, create their own streams connected to files on disk. To use these features, you must

```
#include <fstream>
```

at the top of your file.

To link a stream to a file, we first need a filename. For input streams, this file must already exist on the disk. For output streams, the file may be created by the program. The file name is just a character string, which may be initialised as a constant or typed in at the computer or even created within the program. The examples will consider the filename to be a constant.

Thus
```
     const infilename[] = "input.dat";
```

```
const outfilename[] = "output.dat";
```

would suffice. Note the use of an extention of `.dat` on the file names, so that we avoid accidentally messing with program files.

To link the file to a stream, we first declare an identifier (like `cin`) for the input stream using

```
ifstream ins;
```

where `ifstream` is a new type created in C++ as part of the `fstream` library. `ins` could be any identifier.

To declare an output stream, we use

```
ofstream outs;
```

Now that we have the variable names for the streams, we need to link these streams to the files using

```
ins.open(infilename);
```

and

```
outs.open(outfilename);
```

Note the use of the prefixes `ins.` and `outs.` (our stream names), just as in earlier stream functions like `get()`, `put()`, `good()`, `eof()` and `setw()`. In fact, all these functions are available for file streams, as are the input and output operators >> and << and `endl`.

Here are some more functions, along with some uses for the functions mentioned earlier.

When we attempt to open a file (i.e. attach it to a stream), something might go wrong. For example an input file may not exist. The function `good()` will be false if the open failed. We could also use

```
int fs.fail();
```

which is true if any action on the stream `fs` has just failed (sort of the negative of `good()`).

The function

```
int fs.close();
```

disconnects the stream `fs` from its file.

Here's an example

```
ifstream df;

df.open("input.dat");
if (df.fail())
     cerr << "Could not open input.dat.  Sorry\n";
else
{
     int sum, x;
     df >> x;
     while (!df.eof())
     {
          sum += x;
          df >> x;
     }
     df.close();
     cout << "The sum of the values in file = "
                << sum << endl;
}
```

You should ensure that there is a newline at the end of an input file to avoid problems getting the last piece of information from the file.

Further input/output will be covered in later courses.

12. New data types

One of the strengths of C++ (and to a lesser extent C) is the ability to create your own data types so you are not restricted to just integers, reals and characters. We have already seen a few new data types. The stream types `istream` and `ostream`, and the type `size_t` for measuring the space occupied by variables are created and provided though header files such as `iostream`, `iomanip` and `fstream`. The advantage of C++ is the ability to define operators to handle any new data types. Again, stream operators << and >> are examples.

In CSCI121, you will encounter more about data types. At this time, we introduce three ways in which we can create our own very specific data types enumerated types, aliases and structs. These are all features from C extended in C++.

12.1 Enumerated types

We have seen how we can name constants to allow for program readability. Sometimes we would like to manipulate variables which are limited to a small collection of values often represented by names instead of numeric values.

Consider the case of financial analysis of a business. We want to store the sales figures for the twelve months of the financial year. This could be done by declaring

```
float sales[12];
```

and then associating subscript 0 with July, 1 with August, and so on. But it would not be very self-explanatory for `sales[5]` to represent the sales for December. Wouldn't it be nice if we could say `sales[December]`? Well, we can.

C++ provides a way of creating new types which have, as their values, a small collection of identifiers. For example, we could say

```
enum month {July, August, Spetember, October, November, December, January,
                February, March, April, May, June};
```

This is **not** declaring a variable name called `month` but a data type. So then we could declare

```
month current;
```

which now creates a variable called `current` which can take on the values `July`, `August`, and so on.

Note that the values that an enumerated type can take on **must** be valid identifiers, as C++ actually is just creating named integer constants. Thus, the above creation of `month` defines 12 `int` constants called `July`, `August`, ... , `June`. These constants can be used anywhere (within the scope of the type) as integer constants such as subscripts. The values that these named constants have are, by default 0, 1, 2, ...(in this case to 11).

If we wish to associate other constants with the values, we can specify what the values are. Or we can set some identifier to a value and allow C++ to increment for further values. For example

```
enum day {sunday=1, monday, tuesday, wednesday, thursday, friday, saturday};
```

sets the seven values to 1 – 7.

The general form of the enumerated type declaration is

```
enum type {enumerator_list};
```

where the list gives the values as either just an identifier or an assignment of the identifer to a constant integer expression.

Another example is

```
enum base {binary=2, octal = 8, decimal = 10, hex = 16, hexadecimal = 16};
```

for specifying the number base. Note the use of the same numerical value for two values of the type.

Again remember that the identifiers are named constants. Using an identifier as a value here eliminates it as an identifier anywhere else within its scope.

What can we do with a variable of an enumerated type? Well, not very much. Their use is mainly for readability, although their values can be used wherever an integer could be used. The major problem is that none of the usual operators can cope with one of our **user-defined** types, as enumerations can be called. However, the casting function `int` can be used to output the underlying integer value.

That is

```
     enum eye_colour {blue, brown, green};

     eye_colour colour;

     colour = green;                      // ok
     cout << colour << endl;              // this is not
     cout << int(colour) << endl;    // this is - output is 2
```

The reverse cast is also available, that is

```
     eye_colour(1)
```

would be the value `brown`.

If we wanted to use an enumerated type in a for loop, for example

```
     for (current=July; current <= June; current++}
```

the assignment works, the comparison works, but the increment doesn't. Why?

But we could write a function

```
month next(month x)
{
     return month(int(x)+1);
}
```

and then write

```
     for (current=July; current <= June; current=next(current)}
```

In C++ you can even overload the usual operators such as ++ and << to handle data types of your own, but this is beyond the scope of the current subject.

12.2 New data types by alias

Another way of creating new data types is by using an alias for an existing type. The statement

```
     typedef old_type alias_type;
```

allows the programmer to use *`alias_type`* anywhere that *`old_type`* could be used, including in operations and functions. This might not sound very useful, but again one of its uses is for readability of programs. For example, if a program is going to manipulate many variables which are to contain money amounts, we could specify

```
     typedef double dollars;
     dollars salary, tax, house_payment;
```

thereby indicating those variables which should always be output with 2 decimal places.

A much more useful example is the fact that the *`alias_type`* can include array indicators, so that

```
     typedef char line[80];
```

specifies that the type `line` is an array of 80 characters.

```
     typedef double monthly[12];
```

creates the type `monthly` which is 12 `double` values.

Thus

```
     line first, second;
     monthly income, budget;
```

declare four arrays. Even constant types can be created as in

```
     typedef const char message[];
```

For example

```
void error_message(message msg)
{
     cout << "ERROR: " << msg << endl;
}
```

could be called by

```
     message data_err = "Wrong data"
     error_message(data_err);
```

12.3 Structs

In the last section, we introduced the concept of a (user-defined) type which could contain a collection of like values, as in

```
     typedef monthly float[12];
     monthly salary;
```

where `salary` is a collection of 12 `float` values.

C++ provides a method of creating a type to store a collection of unlike values. For example, a person's name could be described as having three components: surname, first name and initial. We might use a character array of size 40 for surname, another of size 20 for first name and a character variable for initial. To create a type to store such a collection we write

```
     struct name
     {
          char surname[40];
          char firstname[20];
          char initial;
     };
```

Here are some other examples.

```
     struct complex
     {
          float real_part, imaginary_part;
     };

     struct customer
     {
          name cname;
          char address[80];
          gender sex;
     };
```

where `gender` could be an enumerated type defined as

```
     enum gender {female, male};
```

We can then declare variables of these types, such as

```
     name student;
     complex z;
```

The components within a `struct` are called **members**. We know that the access to individual components of an array is by subscript. For `struct`s, we access its members using the notation

```
     structvar.partname
```

(in the same way that those prefixes were used for stream functions).

Thus for variable `student` declared above as type `name`, the three members are

```
     student.surname
     student.firstname
```

and

```
     student.initial
```

while

```
     customer current;
```

has parts like

```
     current.cname
     current.sex
```

and

```
    current.cname.surname
```
down to
```
    current.cname.surname[0]
```

12.3.1 Struct manipulation

The only operations defined for a `struct` are assignment, being passed as a function argument and being returned as a function value. Thus

```
    var1 = var2
```

is legitimate if both *`var1`* and *`var2`* are of the same `struct` type.

Initialisation of a `struct` type follows a similar procedure as for arrays. For example, using the type `customer` defined in the last section we could then declare

```
    customer cust1 = {{"Smith","Fred",'C'},
                          "International House", male};
```

Note the use of nested `{` and `}` to show the sub-groups. The inner ones are only for readability and could be removed.

To input or output a `struct`, we need to access the individual basic data types, not just the members. For example, to output `cust1` would require

```
    cout << "Customer Name : " << cust1.cname.surname
         << ", " << cust1.cname.firstname << ' '
         << cust1.cname.initial << ".\nAddress: "
         << cust1.address << "\nGender: "
         << (cust1.sex ? "Male":"Female") << endl;
```

and would yield

```
Customer Name: Smith, Fred C.
Address: International House
Gender: Male
```

Overloading of operators for `struct` types, and the introduction of a similar construction, called the **`class`**, will be left for CSCI121.

12.4 Arrays of user-defined types

Once we have declared a user-defined type, we can then create arrays of these types. Thus we could have an array of type `customer` or of type `gender`. These are treated in the same ways as arrays of the basic types. Thus, we can reference them using a subscript or pass the name of the array to mean a pass by reference. We still cannot input or output a whole array, nor receive arrays as return values of functions.

12.5 Arrays of arrays - Multidimensional arrays

Many applications in the real world involve grids of values. For example, suppose we wanted to write a program to play chess. We could define the pieces by a `struct` consisting of two `enum` variables.

```
    enum colour {black, white};
    enum rank {king, queen, bishop, knight, rook, pawn};

    struct piece
    {
         colour c;
         rank r;
    };
```

The playing board consists of 8 rows of 8 squares. We could define a row as

```
    typedef piece row[8];
```

and then define the board as

```
    row board[8];
```

This means that the board is made up of 8 rows called `board`, each consisting of 8 `pieces`.
Playing pieces in row 3, numbered from top to bottom starting with row 0, are represented by

```
    board[3]
```

The piece in column 2 (again numbered from 0) of that row is represented by

```
board[3][2]
```

that is, the 2nd element of the 3rd row.

The colour of the piece found there would be

```
board[3][2].c
```

and the rank of the piece is

```
board[3][2].r
```

C++ provides another mechanism for referencing such **rectangular arrays**. We just add the number of columns to the variable name on declaration. So we could say

```
piece square[8][8];
```

and reference the same piece above as

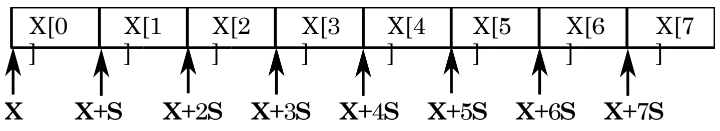
```
square[3][2]
```

To visualise how C++ arranges memory, consider the **one-dimensional** array

```
int X[8];
```

and that the following diagram illustrates the 4 bytes occupied by one `int` in the array:

Let's just say that one element occupies **S** bytes. Arrays are always allocated consecutive memory locations. So the array `X` would occupy 8 consecutive pieces of memory. That is 8**S** bytes.
C++ knows where to find any of the elements of the array. The location of the first element, X[0], is at the beginning of the 8**S** bytes, called the **address** of `X`, which we'll write as **X**. The address of `X[1]` is **X**+ **S**. `X[2]` is at **X** + 2**S**. `X[i]` is at **X** + i**S**.



Now suppose that X is actually a **two-dimensional** array, declared as

```
int X[5][8];
```

Thus there are 40**S** bytes used to represent the 40 elements of the array. To find where an individual array element `X[i][j]` is found, just realise that this doubly subscripted array is like an array of arrays 5 rows of 8 elements. Thus the beginning of the array, address **X** is now the address of `X[0][0]`, **X** + **S** is the address of `X[0][1]`, and the location of `X[1][0]` is after a full row of the array, namely 8**S** bytes. Thus

the address of `X[i][j]` is **X** + (i * rowlength + j) * **S**

The expression `X[i]` is the address of the `i`th row, which is **X** + i*rowlength***S**. Thus `X[i][j]` is the `i`th element of the array with address `X[i]`.

So the compiler has to know the **dimension** of the second subscript (the number of columns) to find the correct entry. It **doesn't** need to know the number of rows.

If you are wondering why **we** need to understand this, consider a function to calculate the sum of all the elements in an array like `X`.

```
int sum_array(int arr[][8], int nrows)
{
      int sum = 0, i, j;

      for (i=0;i<nrows;i++)
            for (j=0;j<8;j++)
                  sum += arr[i][j];
      return sum;
}
```

Note that the formal argument must specify the column dimension, so that the compiler can calculate the location of `arr[i][j]`. This severely restricts C++'s implementation of matrix arithmetic. (There are ways around the problem.)

Initialisation of two-dimensional arrays follows the same format as for one-dimensional arrays, listing the values by row.
So

```
      int X[3][2] = {1,2,3,4,5,6};
```

assigns `X[0][0]` the value 1, `X[0][1]` as 2 and so on. `X[2][0]` is 5. As for initialising `struct` variables, using braces to surround rows (and sometimes newlines) can make the initialisation clearer.

```
      int X[3][2] = {{1,2},{3,4},{5,6}};
```

Back to the chess problem. The squares have been declared as

```
      piece square[8][8];
```

We can initialise the entire starting layout as

```
      piece startpos[8][8] =
            {{black,rook},{black,knight},{black,bishop},
      // followed by the remaining 61 squares.
```

Using loops may be better. We'd better change the pieces to include an empty square, say by redefining `colour` to include the value `empty` (or include `none` in the ranks, or both). Then

```
      for (j=0;j<8;j++)
      {
            square[0][j].c = square[1][j].c = black;
            square[6][j].c = square[7][j].c = white;
            for (i=2;i<6;i++)
                  square[i][j].c = empty;
            square[1][j].r = square[6][j].r = pawn;
//                and so on for the ranks of the other pieces
      }
```

The number of subscripts can be greater than 2. For example, in a chess game, we might use

```
      piece board[200][8][8];
```

to store 200 consecutive board layouts during a game.

When passing a **multi-dimensional** array to a function, all but the first dimension must be included in the formal argument.

Similarly, all but the first dimension is needed when initialising a multi-dimensional array. C++ will work out the first. For example,

```
      int array[][2] = {1,2,3,4,5,6};
```

would determine that 3 is the needed first dimension, while

```
      char day_name[][10] = {"Sunday","Monday","Tuesday",
                  "Wednesday","Thursday","Friday","Saturday"};
```

means this array is 7 by 10.

13. Recursion II

We saw earlier how functions can call themselves. This is called **direct recursion** if the call to the function is within the function. If a function calls a second function which in turn calls the first function, or perhaps calls a third function which calls the first, this is called **indirect recursion**. In either way, we are relying on the function mechanics to keep copies of the return location and the values of local variables at each invocation of a function. For example, consider the following recursive function for calculating the integer power of a double value.

```
double pow(double x, unsigned int n)
{
      if (n == 0)
            return 1;                  // the trivial case
      else
            return x*pow(x,n-1); // the simplifying case
}
```

When this function is invoked for the first time, three pieces of memory are made available: the return location, somewhere for the value of x, somewhere for the value of n.
If the value of n is 0, the value 1 is returned and execution resumed at that saved return location and the memory for the three items is released.
If n is not zero, three new pieces of memory are used for the new return address, x and n-1 (which is n inside the function) and control is passed to the beginning of the function.
And so on.

Note that there will exist a memory location for x (and for n) for each invocation of the function. Which one the function is using at the time is controlled by the stack frame at the moment of starting the function's execution.

As with many recursive functions, the above function can be replaced by an iterative solution.

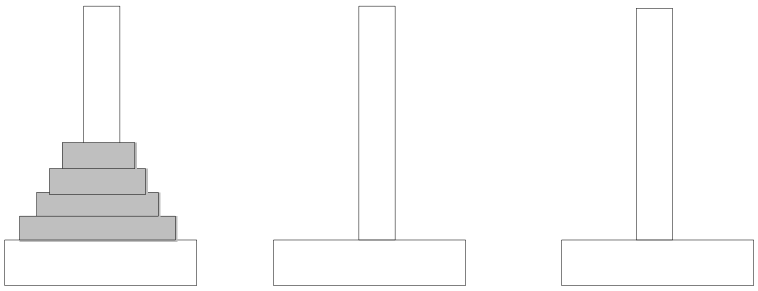
```
double pow(double x, unsigned int n)
{
      double answer = 1;

      while (n > 0)
      {
            answer *= x;
            n--;
      }
      return answer;
}
```

In fact, compilers can often **optimize** the code for a recursive function and remove the recursion in favour of iteration. In this case, recursion is often the more readable description of the function, while the compiler creates the more efficient code. However, optimization is time-consuming, and sometimes writing functions as iterative is better. The function for factorial can be more efficiently written iteratively.

Sometimes, however, a problem begs to be solved recursively. This occurs when the solution to the problem is not inherently obvious, but a way of making the problem simpler is. Consider the classic problem called **The Towers of Hanoi**.

13.1 The Towers of Hanoi



We have three pegs, the first of which has n rings on it, of prorgressively decreasing diameter. The other two pegs are empty.

The problem is to transfer the rings, one at a time, from the first peg to the third using the second as a transfer point ensuring that at no time a ring is on top of another ring of smaller diameter.

The simple case is for one ring. If there is one ring on peg 1, then move it to peg 3. This is the case to use, provided we can move all the other rings to peg2 first.

That problem is moving n-1 pegs from peg1 to peg2 (using peg3 as transfer) is just another variation of the original problem. A simpler one!

Once we have the n-1 rings on peg2, we can move the bottom ring from peg1 to peg3. And then move the n-1 rings from peg2 to peg3 (same problem as the latter one).

Here is a recursive function to solve the problem.

```
void towers(int n, int source, int destn, int transfer)
{
     if (n == 0) return;
     towers(n-1,source,transfer,destn);
     cout << "Move ring " << n << " from peg " << source <<
                " to peg " << destn <<".\n";
     towers(n-1,transfer,destn,source);
}
```

A call of

```
     towers(4,1,3,2);
```

results in the following output:

```
Move ring 1 from peg 1 to peg 2.
Move ring 2 from peg 1 to peg 3.
Move ring 1 from peg 2 to peg 3.
Move ring 3 from peg 1 to peg 2.
Move ring 1 from peg 3 to peg 1.
Move ring 2 from peg 3 to peg 2.
Move ring 1 from peg 1 to peg 2.
Move ring 4 from peg 1 to peg 3.
Move ring 1 from peg 2 to peg 3.
Move ring 2 from peg 2 to peg 1.
Move ring 1 from peg 3 to peg 1.
Move ring 3 from peg 2 to peg 3.
Move ring 1 from peg 1 to peg 2.
Move ring 2 from peg 1 to peg 3.
Move ring 1 from peg 2 to peg 3.
```