The `for` loop statement

- Imagine we put the following code in `main` in one of the `roomba` examples:

```
int myCount;

for(myCount = 1 ; myCount <= 5 ; myCount++)
{

    goNorth();
    goEast();

}
```

- This would make the roomba go 5 steps north and five steps east.

The `for` loop statement

- General structure:

```
for(<start>; <true or false> ; <change>)
{

    <some instructions>

}
```

- This works as follows

FOR LOOPS, INPUT AND FILE HANDLING

Today

- The `for` statement
- How to read data in from a file.
- How to write data out to a file.

These notes go along with the *patient record* examples.

The `for` loop statement

- At the start of the loop, the instruction in `<start>` is carried out.
- We usually use this to set the value of a counter.
- Then `<true or false>` is tested to see if it is true or false.
- This is usually a test on the counter.
- If it is false, the program will skip to the } that marks the end of the control structure.
- If it is true the `<some instructions>` are executed.
- Once they are done, the instruction in `<change>` is executed.
- This is usually something that changes the value of the counter.
- Then `<true or false>` is tested again.
- Thus `<some instructions>` will be repeatedly executed until `<true or false>` becomes false.

Examples

- Let's go back to our initial example:

```
int myCount;

for(myCount = 1 ; myCount <= 5 ; myCount++)
{

    goNorth();
    goEast();

}
```

- This would make the roomba go 5 steps north and five steps east.

- While:

```
int myCount;

for(myCount = 10 ; myCount > 5 ; myCount--)
{

    goNorth();
    goEast();

}
```

would do the same, but with different values of `myCount`.

- What would

```
int myCount;

for(myCount = 2 ; myCount < 8 ; myCount+=2)
{

    goSouth();
    goWest();

}
```

do?

Simple input

- The function `cin` allows us to read input from the keyboard.
- A typical pattern of usage is:

```
cout << "Enter a number" << endl;
cin  >> x;
```

  which first *prompts* the user, then reads the next thing they type into the variable `x`.
- Other examples of using `cin` can be found in the various `roomba` programs on the course website.
- `cin` is the counterpart of `cout`.

Reading from a file

- To read from a file, we have to tell the program three things:
  - That we are going to read from a file.
  - How we will refer to the file inside the program.
  - What the name of the file on the hard drive is.
- We can do those three things using one command:

```
ifstream infile("patient.dat");
```

- The `ifstream` says we are going to read from a file.
- `infile` is the name we are going to use inside the program.
- `patient.dat` is the name of the file on the hard drive.

Using files

- In the same way as we use `cin` to read data from the keyboard, we can read data from files.
- In the same way as we use `cout` to write data to the screen, we can write data to files.
- This allows us to store information on the computer's hard drive, and to use it when we want it without having to type it in each time.

File preliminaries

- To use read and write data to a file, we will make use of some *library functions*.
- To use these functions we need to add:

```
#include <fstream>
```

  at the start of the program.
- We put this in the same place as:

```
#include <iostream>
```

Writing to a file

- To write to a file, we have to tell the program three things:
  - That we are going to write to a file.
  - How we will refer to the file inside the program.
  - What the name of the file on the hard drive is.
- Again we can do those three things using one command:

```
ofstream outfile("patient.dat");
```

or using two commands:

```
ofstream outfile;
outfile.open("patient.dat");
```


- Once we have defined `outfile` as an *output stream*, we can send data to it.
- We use `outfile` much like `cout`.
- Thus:

```
outfile << idNumber;
```

sends the value of the variable `idNumber` to the file.

- Once we have finished reading from the file, we close it:

```
outfile.close();
```

- When writing to a file it is important to close it — if the file isn't closed, the data that we have set to the file might not be stored in it.

- Alternatively we can write this as two commands:

```
ifstream infile;
infile.open("patient.dat");
```


- Now that we have defined `infile` as an *input stream*, we can read data from it.
- We use `infile` much like `cin`.
- Thus:

```
infile >> idNumber;
```

reads the next integer from the file into the variable `idNumber`

- Once we have finished reading from the file, we close it:

```
infile.close();
```

- Finally, we have have:

```
outfile.open("patient.dat", ios::noreplace);
```

  which will fail to open the file if it does already exist.
- `noreplace` is thus the dual of `nocreate`.
- There are also modes for input files.
- We have:

```
ifstream myfile;
myfile.open("patient.dat", ios::in);
```

  which will open the file for input.

Summary

- This lecture started with `for` loops.
- Then we looked at input from the keyboard.
- Lastly we started on simple file handling.
- In particular, we looked at:
  - reading data in from; and
  - writing data out to

  simple sequential files.

File open modes

- When we open a file for writing, the computer discards any information that is in the file.
- This is not always what we want to do.
- We can control what happens by specifying the *file open mode*.
- For example, instead of:

```
ofstream outfile;
outfile.open("patient.dat");
```

  we can have:

```
ofstream outfile;
outfile.open("patient.dat", ios::app);
```

  which will write new output to the end of the file.

- There are other options.

```
outfile.open("patient.dat", ios::trunc);
```

  will discard any information in the file.

```
outfile.open("patient.dat", ios::out);
```

  will open the file for output, and is just another way to do:

```
outfile.open("patient.dat");
```

- We also have:

```
outfile.open("patient.dat", ios::nocreate);
```

  which will fail to open the file if it doesn't already exist.