1A C++ coursework, Michaelmas Term (with extras)

This document makes no attempt to exhaustively cover C++ - the recommended book by Deitel and Deitel is 1300 pages long and even that is incomplete.

The first few sections introduce just enough C++ for you to write small programs and run them. The idea is to give you the confidence to learn more yourself - like learning to snowplough as the first stage when skiing, or learning not to be scared of the water when learning to swim. Once you can compile and run programs, you can copy example source code and experiment with it to learn how it works. Like skiing and swimming, you can only learn programming by doing it. If you need more exercises like the early ones, look in the More exercises section. The later sections revisit topics, adding more detail.

The C++ language and the methodologies described here will appear strange if you have never written a computer program. Don't attempt to "understand" everything from first principles. More explanation will be given later here or in the CUED Tutorial Guide to C++ Programming. For now, concentrate on learning how to use the language. The document includes some quick tests so that you can check your understanding, and some sections of extra information that you can show or hide. If you want to hide all the extra sections, click on this No-extras button. Use Ctrl + and Ctrl - to adjust the text size.

The course comprises 6 timetabled sessions, each beginning with a mini-lecture.

<table>
<thead>
<tr>
<th>Session</th>
<th>Themes</th>
<th>Exercises</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction (programs, variables, etc)</td>
<td>1. Adding, 2. Times table</td>
</tr>
<tr>
<td>3</td>
<td>Math functions, number representation</td>
<td>5. Word lengths, 6. Accuracy, 7. Pi</td>
</tr>
<tr>
<td>4</td>
<td>Data abstraction, structs, enums etc.</td>
<td>8. Monopoly, 9. A phone directory</td>
</tr>
<tr>
<td>6</td>
<td>Algorithmic complexity</td>
<td>12. Measuring program speed</td>
</tr>
</tbody>
</table>

Work through the document at your own speed. Getting ahead of this schedule is fine, but you shouldn't fall far behind it. When you've finished exercises 1-4 get a demonstrator to mark your work (4 marks). Try to get at least that far on your first day of programming.

When you've had exercises 1-4 marked, continue with the other exercises. If you don't finish exercise 5 on day 1, finish it before day 2. Contact Tim Love (tl136) for help. When you've finished exercises 5-9, get them marked (5 marks), then finish exercises 10-12 before the final marking (3 marks). If you finish early, you're strongly advised to try some More exercises.

Start sessions in the DPO by clicking on the Applications icon at the bottom of the screen and then clicking on the All CUED Applications option, then the "1st Year" option and then "Start 1AComputing". This will put some icons on the screen for you, and give you a folder called 1AC++Examples full of example source code. It also creates a folder called 1AComputing, a good place to store any course-related files.

If you want to work from home, see our Installing C++ compilers page.

In the course of this work your screen might become rather cluttered. The Window management section of the New User Guide has some useful tips.

Variables

Variables are places to store things. The line
int num;

creates a variable called `num` in which an integer can be stored. Note the final semi-colon - in C++, semi-colons are a little like full-stops at the end of English sentences. You can also have variables that store a

- `float` ("floating point" number - a real number)
- `char` (character)
- `string` (text)

etc. To set an existing variable to a value, use an `=` sign. E.g.

```plaintext
num=5;
```

or

```plaintext
num=num+1;
```

The latter line might look a bit strange at first sight, but it isn't saying that the LHS is the same as the RHS. It's an assignment - it's setting the LHS to the value of the RHS - i.e. adding 1 to `num`.

You can create and set a variable in one line. E.g.

```plaintext
float num=5.1;
```

C++ is fussy about variable names - they can't have spaces or dots in them, nor can they begin with a digit. It distinguishes between upper and lower case characters - `num` and `Num` are different variables. It's also fussy about the type of the variable - you can't put text into an integer variable, for example.

You can't do

```plaintext
int i="hello";
```

because text can't be stored in an integer variable. You can't even do

```plaintext
int i="48";
```

You can try putting a floating point number into an integer variable. If you try

```plaintext
int i=4.8;
```

it will set `i` to 4.

If you want to make a variable "read-only" you can use `const` so that its value can't change once it's been set. For example, the following line fixes the value of `x`
const int x = 42;

If a variable is never going to have a negative value, you can specify it as unsigned. For example, num in the example below will never be negative

unsigned int num;

You can use base 8 (octal) notation by beginning a number with a zero, or base 16 (hexadecimal) notation by beginning a number with zero followed by x. For example, the following lines both set variables to the base 10 value 20

int a = 024; int b = 0x14;

You can't have 2 variables with the same name in the same context. Here are some examples.

```cpp
int main() {
    int i;
    int i;
}
```

The compiler reports error: redeclaration of 'int i'

```cpp
int main() {
    int i;
    float i;
}
```

The compiler reports error: conflicting declaration 'float i'

```cpp
int fun() {
    int i;
}
```

```cpp
int main() {
    int i;
}
```

No problem. Each i variable is inside a function. There's no clash.

```cpp
int main() {
    int i;

    for (int i=0;i<4;i++) {
        i;
    }
}
```

Legal. The i created inside the for is used inside the loop. Outside the loop, the other i variable in used. C++ is happy with that - there's no ambiguity.

In this document we use the camelCase convention for variable names. Names will begin with a lower-case letter and each new word
Strings and Characters

Strings are sequences of characters. If you add 2 strings using +, the result will be that the 2nd string is appended to the 1st. Strings are more sophisticated than simple data like ints and floats (technically speaking, a string is an object). They have extra functionality associated with them. For example, if you want to find the length of a string s, you can use s.length(). Here’s an example of appending to, then finding the length of, a string:

```cpp
string s="hello";
s=s+" world";
int l=s.length();
```

To find a particular character in a string (the 3rd, for example) use this method:

```cpp
string s="hello";
char thirdCharacter=s[2];
```

Note that the numbering of the characters starts at 0.

Whereas strings have double quotes around them, characters have single quotes, so to create a character variable and set it to x you need to do:

```cpp
char c='x';
```

Objects are widely used in C++ (it’s said to be an ‘Object-orientated’ language). Later you’ll be creating your own types of objects. For now, you’ll be using off-the-shelf ones like strings. length is one of many facilities available - there’s also find, erase, etc.

Output

Use cout (short for “console output”) to print to the screen. Before each thing you print out you need to have << (having 2 “less-than” symbols together like this is nothing to do with the mathematical “less-than” operator) . So

```cpp
cout << 5;
```

prints out a 5, and

```cpp
cout << num;
```

prints out the value of the num variable. If you want to print text out, put double-quotes around it - e.g.

```cpp
cout << "hello";
```

prints hello. To end a line and start a new one, use the special symbol endl.
You can print several things out at once, so

```cpp
int num=5;
cout << "The value of num=\" << num << endl;
```

prints

```
The value of num=5
```

### Putting it all together [back to contents]

All the examples so far have been fragments. Now you're going to write complete programs. All the C++ programs that you're likely to write will need the following framework. The Input/Output and string functionality is not actually part of the core language. To use it you need the following lines at the start of your code.

```cpp
#include <iostream>
#include <string>
using namespace std;
```

In C++ a **function** is a runnable bit of code that has a name. The code might calculate a value (like a function in mathematics does) but it might just perform a task (like printing something to the screen). Every C++ program has a function called **main**. When the program is started, the main function is run first. So your program needs a **main** function which will look like this.

```cpp
int main() {
...
}
```

Don't worry for now what this all means. Just remember that all your programs will probably need lines like these.

Now we'll write a minimal program. At the moment your **1AComputing** folder is nearly empty (you'll need the **secretmessage** file soon). In the **File** menu of the **1AComputing - File Browser** window pick the **Create Document** option and create a file called **program1.cc**. Drop the file into the Geany icon on the desktop and you'll get the following window.
Geany 0.20.
sers/tpl/1AComputing/program1.cc opened(1).

geany has lots of features to help with writing C++ programs, including an editor. Type (or copy-paste) the following text into it. It’s a complete program with some initial setting-up lines and a main function containing some code. Note that // and anything after it on a line is ignored by the compiler - you can add comments to your programs this way to remind yourself about how they work.

```cpp
#include <iostream> // We want to use input/output functions
#include <string> // We want to use strings
using namespace std; // We want to use the standard versions of
// the above functions.

// My first program! This is the main function
int main() {
    int i=3; // Create an integer variable and set it to 3
    // print the value of i and end the line
    cout << "i has the value " << i << endl;
}
```

In the Build menu, choose the Build option (or use the icon). This will try to compile your code. If you’ve made no typing mistakes then you’ll see “Compilation finished successfully” and a file called program1 will be created, which is in a form that the computer’s chip can understand. You’ll be able to click on the icon to run this program. If it prints out i has the value 3 you’ve produced your first program!

Note that geany

- colour-codes the program to make it more readable. If you don’t get colours it’s because you haven’t named the file with a “.cc” suffix - geany expects C++ filenames to have that suffix
- shows line numbers but those line numbers aren’t in the C++ source file.
- saves the file automatically whenever you do a build

Though the compiler doesn’t care about the layout, you should make your code easy to read by copying the layout style of the provided code. In particular, use indentation consistently.

[hide extra information]

On the CUED machines, Linux machines and some Macs you can work from a terminal window instead. Use an editor like emacs or gedit that can easily produce text-only files. To compile and then run a program called program1.cc, try typing

```
g++ -o program1 program1.cc
./program1
```
Errors and Warnings

You may not get everything right first time. Don't be worried by the number of error messages - the compiler is trying to give as much help as it can. You can get a lot of error messages from one mistake so just look at the first error message. Clicking on the error message in geany will move the editor's cursor to the corresponding line. Often, the compiler will report that the error is in a line just after where the mistake is. If you cannot spot the mistake straight away, look at the lines immediately preceding the reported line.

The most common errors at this stage will be due to undeclared variables or variables incorrectly declared, and missing or incorrect punctuation. Check to see that brackets match up, and check your spelling. For example, if you have a source file called program2.cc with

```cpp
ant i;
```

instead of

```cpp
int i;
```

on line 3 our compiler might give the message

```
program2.cc:3: error: ant does not name a type
```

This message tells you

- the filename (program2.cc)
- the line number where the compiler had trouble (line 3)
- a description of the problem.

It may not tell you exactly what's wrong, but it's a clue. Sometimes the compiler doesn't give very helpful messages at all. E.g. if you write

```cpp
cin >> endl;
```

instead of

```cpp
cout << endl;
```

the compiler will give you a page of obscure messages like this

```
/usr/include/c++/4.3/iostream.tcc:858: note: std::basic_istream<_CharT, _Traits>&
std::operator>>(std::basic_istream<_CharT, _Traits>&, _CharT&)
[with _CharT = char, _Traits = std::char_traits<char>]
```

Don't panic. All you can do is look at the first line number that's mentioned in the list of errors, and study that line of code.

When you think you've identified the trouble, correct it, save the file and build again.

Even if your code is legal and builds without error, your code may do the wrong thing - perhaps...
because you've put a '+' instead of a '-'. One of the most effective things to do in this situation is to use `cout` to print out the values of certain variables to help you diagnose where the problem is. Don't just passively stare at your code - make it print out clues for you.

Many common bugs are explained on our C++ Frequently Asked Questions page. Look there first before asking a demonstrator for help. Many more tips are in the Troubleshooting section.

Sometimes compilers will warn you about something that's legal but suspicious. It's worth worrying about such warnings, though they might not be a problem. For instance, if you create a variable called `cnew` and don't use it, the compiler might report

```
warning: unused variable 'cnew'
```

[C++ was built for speed and flexibility, not for safety or ease of use. Its predecessor, a language called C, has been compared to a go-kart with a rocket motor. C++ has safety belts and a dashboard of instruments, but they're optional extras. You are responsible for writing safe code - the compiler won't help you. Train yourself to use safer alternatives when there's a choice. Program pessimistically.]

Your problems when writing a program are likely occur in all phases of writing

- **Design** - Before you grapple with C++ (or any other language) you need to know what you want to say. You need a plan - an algorithm - to solve the problem. If you don't know how to determine HCFs you won't be able to write a program to calculate them.

- **Coding** - Even if you know in principle what you want to do, C++ is an unforgiving language. Unless you write legal C++ the compiler won't produce a runnable program. The compiler's error messages should at least indicate where the problem is even if the nature of the problem isn't clear. See the Compiling section of our FAQ (Frequently Asked Questions) for some tips.

- **Execution** - Even if the compiler produces a program for you, the program might crash, freeze, or give the wrong answer. Some bugs produce erratic symptoms - your program might sometimes work and sometimes not.

Check that variables have the expected values by printing the values out. Check that particular lines of code are being reached by putting

```
cout << "Got Here!" << endl;
```

by the line. Give the program a dry-run - simulate on paper what the computer will do. See the Executing section of our FAQ (Frequently Asked Questions) for some tips.

Input  

Use `cin` (short for "console input") to get values from the keyboard into variables. Before each thing you input you need to have `>>`. The variables need to be created beforehand. E.g.

```cpp
int num;
cin >> num;
```

will wait for the user to type something (they have to press the Return key after). If they type an integer, it will be stored in the `num` variable.

You can input several things on one line, like so

```cpp
int num, angle, weight;
cin >> num >> angle >> weight;
```

A common error is to try to input 2 numbers by doing


```cpp
    cin >> num1, num2;
```

Though this is legal C++, only 1 number will be given a value. You need a >> symbol for each variable.

Note that in real programs you have to contend with users who'll type a word instead of a number, or users who type a negative weight.

**Exercise 1 - Adding**

You now know enough to write your own programs. Use *geany*'s "New" option to create a new file. Save it as *adding.cc* in your 1AComputing folder. You're going to write a program that prints the sum of 2 integers typed in by the user.

I suggest you start by taking a copy of *program1.cc* and removing the contents of the *main* function. Your function needs to:

- Create 2 integer variables
- Ask the user to type in 2 integers
- Use *cin* to read the values into your variables
- Print an output line looking rather like this

```
The sum of 6 and 73 is 79
```

Write the code to do this now.

**Decisions**

Use the *if* keyword. Here's an example

```cpp
    if(num<5) {
        cout << "num is less than 5" << endl;
    }
```

The curly brackets are used to show which code is run if the condition is true. You can have many lines of code within the curly brackets. Instead of using `<` (meaning 'less than') to compare values you can use:

- `<=` (meaning 'is less than or equal to')
- `>` (meaning 'is greater than')
- `>=` (meaning 'is greater than or equal to')
- `!=` (meaning 'isn't equal to')
- `==` (meaning 'is equal to').

A common error in C++ is to use `=` to check for equality. If you do this on our system with *geany*, the compiler will say:

```
warning: suggest parentheses around assignment used as truth value
```

but many compilers won't say anything. Train yourself to use `==` when making comparisons (some languages use 3 equals signs so count your blessings). Be careful to avoid mixed-type comparisons - if you compare a floating point number with an integer the equality tests may not work as expected.

You can use *else* in combination with *if* - e.g.

```cpp
    if(num<5) {
        cout << "num is less than 5" << endl;
    } else {
```

cout << "num is greater than or equal to 5" << endl;
}

You can combine comparisons using boolean logic. Suppose you want to run a line of code if `num` is between 3 and 5. The following doesn't work as expected though the compiler won't complain!

```cpp
if (3<num<5)
...
```

Instead you need to use

```cpp
if (3<num and num<5)
...
```

As well as `and`, the C++ language understands `or` and `not`.

Don't put a semi-colon straight after `if(...)`

[hide extra information]

3<num<5 is likely to be treated as `(3<num)<5`. The first expression in brackets will be true or false. In this context, true and false will have the values 1 and 0 respectively, so the original expression will reduce to `1<5 or 0<5`. Whichever, the result is `true`.

You need to take care when comparing floating point numbers. Look at the following code. What will it print out when run?

```cpp
#include <iostream>
using namespace std;

int main() {
    float number=0.0;
    while (number!=0.2) {
        cout << "Number=" << number  << endl;
        number=number+0.1;
    }
}
```

If you run this program it might run for a very long time. `0.0 + 0.1 + 0.1` doesn't equal exactly `0.2` because of the way computers store numbers. With real numbers on computers you need to work within a tolerance (like an engineer) rather than assume perfect accuracy.

**Arithmetic**

Use `+` (add), `-` (subtract), `*` (multiply), `/` (divide), and `%` (modulus - i.e. getting the remainder in integer division). Note that there's no operator for exponentiation (in particular, `3^2` doesn't produce `9`). The order of execution of mathematical operations is governed by rules of precedence similar to those of algebraic expressions. Parentheses are always evaluated first, followed by multiplication, division and modulus operations. Addition and subtraction are last. The best thing, however, is to use parentheses (brackets) instead of trying to remember the rules.

You can't omit `*` the way you can with algebra. For example you have to use `2*y` to find out what 2 times `y` is. `2y` is illegal.

Although addition, subtraction and multiplication are the same for both `integers` and `floats`, division is different. If you write

```cpp
float a=13.0, b=4.0, result;
result = a/b;
```

then real division is performed and `result` becomes `3.25`. You get a different result if the operands are defined as integers:
```cpp
int a=13, b=4;
float result;
result = a/b;
```

The variable `result` is assigned the integer value 3 because in C++, arithmetic performed purely with integers produces an integer as output. If at least one of the numbers is a real, the result will be a real. This explains why later in this document you'll sometimes see `2.0` being used instead of `2` - it forces real division to be done.

**[hide extra information]**

There are about 17 levels of operator precedence in C++. `^` is the exclusive-or operator (in the 11th level of precedence), so `3^2` is 1. You'll see later how to raise a number to a power.

What is the result of doing `1/0`? What is `0/0`? Try them out and you'll see.

### While Loops

For repetitive tasks, use loops. Easiest is the **while** loop - code that repeatedly runs *while* some condition is true. Here's an example:

```cpp
int num=1;
while (num<11) {
    cout << num << endl;
    num=num+1;
}
```

When the computer runs this particular **while** loop, it continues printing `num` and adding one to it while `num` is less than 11, so it prints the integers from 1 to 10.

The indentation of the lines isn't necessary, but it makes the code easier for humans to read, and helps you match up opening and closing braces.

When you write a loop, always make sure that it will eventually stop cycling round, otherwise your program might appear to "freeze". Without the `num=num+1` line in this example, `num` would always be 1 and the loop would cycle forever. If your program does get stuck in a loop like this, use **geany**'s **Stop** button to kill the program.

**Don't** put a semi-colon straight after **while(...)**

**[hide extra information]**

- **while** loops are useful if you don't know beforehand how many times the loop will need to be run. Suppose you want the user to type a number in, and the number needs to be greater than 0. Here's a way to do it:

```cpp
float num=0;
while (num <= 0) {
    cout << "Type a number greater than 0 and press the Return key:";
    cin >> num;
}
```

This loop will go round and round while `num` has an invalid value.

- There's also a **do ... while** loop that does the check at the end of the loop rather than the beginning. The code into such a loop will always be run at least once. Here's an example:

```cpp
int num=1;
do {
    cout << num << endl;
    num=num+1;
}
```

while (num<12);

Exercise 2 - Times Table

Use geany’s “New” option to create a new file. Save it as timetable.cc in your 1AComputing folder. You're going to use a while loop to print out the first 10 entries in the 6 times table - i.e.

1x6=6
2x6=12
...
10x6=60

If you put the while loop example inside a main routine like the one above, your program will nearly be finished - all you need to do is change the cout line so that it not only prints the variable that goes from 1 to 10, but the rest of the line too. Some of the rest of the line doesn't change. The final number does, but it can be expressed in terms of the first number on the line.

Functions

As we mentioned earlier, in C++ a function is a runnable bit of code that has a name. Most programs have many functions. Now you're going to write your own ones. First we'll produce a times table (the 7 times table this time) using functions. Here's a function called timesBy7 that multiplies its input by 7.

```
int timesBy7(int number) {
    return number*7;
}
```

Conceptually, C++ functions are rather like maths functions or functions on your calculator. You can think of them as generating output from their input. They might need 1 number as input (like the square root function) or several, or none. Note that in C++, functions are said to “return” their output back to the thing that asked for them. They can return one thing or nothing. Execution of the function code ends when a return statement is reached, or when the function code ends.

The first line of the function above is compact and contains a lot of information.

- `int` - this is saying that the function is going to calculate an integer value rather than (say) a string.
- `timesBy7` - this is the name of the function
- `(int number)` - this is saying that `timesBy7` needs to be given an integer as input, and inside this function the integer is going to be known as `number`. Some functions don't need any inputs. Other functions might need many inputs. This function needs exactly one integer.

If we wanted to use the function to store the result of 9*7 we could just do

```
int a=timesBy7(9);
```

Here's a little program that uses this function to display some multiples of 7. Though short, it illustrates what you need to do when writing your own functions. From now on, just about all your programs in every computing language you learn will use functions, so study this example carefully

```
#include <iostream>
#include <string>
using namespace std;

// This function multiples the given number by 7 and returns the result
int timesBy7(int number) {
```
return number*7;
}

int main()
int num=1;
while (num<11) {
    cout << timesBy7(num) << endl;
    num=num+1;
}
}

This file contains 2 functions. Like all C++ programs, execution begins at the main function. From the main function, timesBy7 is called. It needs to be given an integer (in this case num) as input. In this example its output is immediately printed out. You can call it in other ways - for example, you could save its output into a variable called answer then print answer.

This file contains 2 functions. Like all C++ programs, execution begins at the main function. From the main function, timesBy7 is called. It needs to be given an integer (in this case num) as input. In this example its output is immediately printed out. You can call it in other ways - for example, you could save its output into a variable called answer then print answer.

... int main() {
    int num=1;
    int answer;
    while (num<11) {
        answer=timesBy7(num);
        cout << answer << endl;
        num=num+1;
    }
}

Note that the timesBy7 function appears in the file before it's called. That's because the C++ compiler doesn't like calling a function if it doesn't know about it first. In fact, it doesn't need to know everything about the function, just the information contained on its first line. That first line can be used by itself (with a semi-colon at the end) to “summarise” the function. It's called a prototype.

It's often convenient to have the function prototypes at the top of the file, and the full function code elsewhere so that's the style we'll adopt from now on.

We're now going to write a program with a function that will tell us whether numbers are even or odd. The program's output will be

0 is even
1 is odd
2 is even
3 is odd
...
10 is even

In the main function below we set i to 0, 1, ... 10 checking each time to see if i is even. The code uses if and else which are quite easy to understand, I hope. You can read the “if” line as saying “if i is even, then do ...”. This code fragment assumes that C++ has a function called is_even which returns true or false.

int main()
    int i=0;
    while(i<11) {
        if (is_even(i)) {   // Or you could have if (is_even(i)==true) {
            cout << i << " is even " << endl;
        }
        else {
            cout << i << " is odd " << endl;
        }
        i=i+1;
    }
Read through the code until you understand it. Don't hesitate to trace your finger along the route the computer takes through the code, keeping a note of what value i has.

Self Test 1 - How many times is the "i=i+1;" line reached?

Unfortunately there's no function called is_even so we'll have to write it ourselves. We could do it many ways. Here we'll use the % operator, which gives us the remainder after integer division. If we do number%2 and the answer is 0, 2 divides exactly into the number, so the number is even. We want the is_even function to give us a true/false answer. In C++ there's a type of variable called bool (short for Boolean) which can store such answers. Here's the is_even function.

```cpp
bool is_even(int number) {
    if ((number % 2) == 0) {
        return true;
    } else {
        return false;
    }
}
```

The prototype of this function is

```cpp
bool is_even(int number);
```

We now have nearly all the code. Below on the left is the complete program with the main and is_even functions we've prepared. To the right is an animation showing what happens when the program runs for a few cycles.
The bold text shows the 3 key aspects of writing your own functions:

1. The **declaration** of the function above the main program. The declaration (also known as the **prototype**) tells the compiler about the function and the type of data it requires and will **return** on completion. Note that the declaration **doesn't** make the code run.

2. The **function call** in the main body of the program determines when to branch to the function and how to return the value of the data computed back to the main program. Note that when you call a function you **don't** mention datatypes (like `int`, `bool`, etc) - you **don't** write `is_even(int i)` to call the function.

3. The **definition** of the function. The definition consists of a **header** which specifies how the function will interface with the main program and a **body** which lists the statements to be executed when the function is called.

The animation's green arrow starts in the **main** routine at the stop sign and goes round and round the loop. It jumps to and from the `is_even` function. Inside that routine it sometimes follows the `if` route and sometimes the `else` route, depending on whether the number is even or odd.

Don't worry if you can't keep up with the animation. The important thing to realise is that the code isn't simply run from top of the file to the bottom - some lines are run many times. It's also worth noting that the `i` variable exists only in the **main** function - that's why it keeps appearing and disappearing at the top of the animation. It's described as a **local** variable.

When you create a function, don't give it a name that's already in use. Don't, for example, call it `int` because that's part of the C++ language. Don't call it something general (like `vector`, or `max`) because C++ often uses such names internally.

Function definitions can't be nested. In the example code above, for example, the main function has to be finished with a final curly bracket before the is_even function can be started.

See the C++ Tutorial Guide for more information.

### Exercise 3 - functions

Adapt the previous example so that instead of identifying even numbers it identifies multiples of 4. Give the function a sensible name. Call the program `multiplesof4.cc`.

### Arrays

When you have lots of variables that are related in some way, it's useful to clump them together somehow. **Arrays** offer a way to do this as long as the variables are all of the same type. For example, if you want to store some integer information about each month in an array, you could use

```cpp
int month[12];
```

to reserve space for 12 integers. Inside the computer the memory layout will be something like this:

The integers have the names `month[0]`, `month[1]` ... `month[11]` because array indexing starts at 0. Array items are usually referred to as **elements**. They're stored contiguously in memory. Arrays and loops go well together. To set all the elements of the `month` array to 0, you could do

```cpp
int i=0;
while (i<12) {
    month[i]=0;
    i=i+1;
}
```
This is much shorter than doing

```cpp
type month[12] = 0;
type month[1]=0;
... type month[11]=0;
```

Be careful when using arrays - if you create an array of 5 elements but you use 6 elements, you're using memory that you haven't asked for, memory that might be being used by something else. A crash is likely.

You could initialize the array using `int month[12]={0};` but be aware that 0 is a special case - doing `int month[12]={1};` sets the first element to 1 and all the others to 0.

An array's size is fixed when the array is declared and remains the same throughout the execution of the program. C++ has a newer, more flexible way to do arrays, using `vector`. The notation's more involved though. Here's an example

```cpp
vector<int> month[12];
```

You'll use this in the 2nd year

**Self-test 2**

Look at this code.

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    int x,y;
    y=3;
    x = y*2;
    y=5;
    cout << x << endl;
}
```

1. How many variables are used?
   - 0
   - 1
   - 2
   - 3
   - 4

2. When it's compiled and run, what will it print out?
   - 2
   - 3
   - 5
   - 6
   - 10

3. What is `main`?
   - A variable of type `int`
   - A function that needs 1 integer as input
   - A function that needs integers `x` and `y` as input
   - An array of integers
   - A function that needs no input parameters

---

To be able to use the file-reading facilities you need to add
to the top of your file. Then you can do the following to get a line of text from a file (in this case a file called `secretmessage`) and put it into a string (in this case called `message`).

```cpp
string message;
ifstream fin; // a variable used for storing info about a file.
   // There's nothing special about the name 'fin' -
   // any name would do.
fin.open("secretmessage"); // trying to open the file for reading
// The next line checks if the file's been opened successfully
if(not fin.good()) { // print an error message
    cout << "Couldn't open the secretmessage file."  << endl;
    cout << "It needs to be in the same folder as your program" << endl;
    return(1); // In the main function, this line quits
          // from the whole program
}
// we've managed to open the file. Now we'll read a line
// from the file into the string
getline(fin, message);
```

This code includes a check to see if the file exists. There are many other file-reading facilities, but that's all you'll need for now.

**More about functions**

The functions we've seen so far have 0 or 1 input values and 1 output value, but C++ is more flexible than that.

- Often the purpose of a function is to work out a value and return it, but sometimes a function will perform a task (like printing to the screen) and won't have anything useful to return. In such situations, the function needn't return anything. If a function called `printOut` returns nothing and needs no input, its prototype would be

```cpp
void printOut();
```

where the `void` word means that nothing is returned.

- Functions can have several input values. For example, a function with the prototype

```cpp
float pow(float x, float y);
```

needs to be given 2 floating point numbers. It returns a floating point number too.

The following table shows how C++ represents various types of function diagrams...
<table>
<thead>
<tr>
<th>C++ function prototype and use</th>
<th>Function diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>int fun1(int x);</td>
<td>![Diagram of fun1 and i]</td>
</tr>
<tr>
<td>int i=fun1(7);</td>
<td></td>
</tr>
<tr>
<td>void fun2(int x);</td>
<td>![Diagram of fun2 and side-effects]</td>
</tr>
<tr>
<td>fun2(7);</td>
<td></td>
</tr>
<tr>
<td>int fun3();</td>
<td>![Diagram of fun3 and i]</td>
</tr>
<tr>
<td>int i=fun3();</td>
<td></td>
</tr>
<tr>
<td>int fun(int x);</td>
<td>![Diagram of fun, fun, and i]</td>
</tr>
<tr>
<td>int i=fun(fun(7));</td>
<td></td>
</tr>
</tbody>
</table>

The variables created in a function can only be used in that function - they're local variables. If you have "int i;" in your main function and another "int i;" in another function the 2 i variables will be independent.

Try the "function" teaching aid to get some practice

**Troubleshooting - a checklist**

The longer your programs become, the harder it will be to fix bugs and the more important it will be to write tidy code. Here are some things to check:

- **Variables**
  - Are they created at the right time? Do they have the right initial value?
  - Are they the right type? (int rather than float maybe, or an array instead of a simple value)
  - Are they sensibly named? Are you clear about what each variable's for and what its contents represent?

- **Program Organisation**
  - Do you have a main function? Does it begin with int main()?
  - Have you included the right files? Do you have using namespace std;?
  - You can create variables outside of functions, but is all your code inside functions?
  - Do you know when one function ends and another begins?
  - Do all your blocks of code (in while loops, if .. else constructions, etc) start and end where they're meant to?
  - Have you put any comments in your code?

- **Functions**
  - Do they have the right inputs and outputs?
  - Does the prototype's input/output specification match the function code's specification?
  - Are you calling the function? How do you know? (add cout commands). Remember that when you call a function you need brackets after the name even if the function requires no input values.
If you're using random numbers are you calling `srandom` exactly once?

Arrays and Loops
- Do your loops go round forever?
- Do you go off the end of an array? (If you do, a “segmentation fault” or an “out of range” error may be reported.)
- Did you remember that the 1st item in an array has an index of 0?

Punctuation
- Are you mixing up `=` and `==`?
- Are you using semi-colons correctly?
- `if` and `while` need to be followed by a condition in brackets. Have you added the brackets?

Strategies and workflow
- If the code’s not compiling, make it compile by removing (or commenting-out) problematic lines until it does. Then restore a line or 2 at a time. Re-build after each addition. Print out some variables to see if the code is behaving.
- If you've changed your code but your program's behaviour hasn't changed, maybe you haven't created a new version of the program. Remember, the Compile button doesn't create a new program, but the Build menu item does - F9 is a short-cut for it.
- When the code runs but does the wrong thing, do a dry run - run through the code on paper as if you were the computer, keeping a note of variables' values, and checking that the correct route is taken through the program.
- Read through the C++ Frequently Asked Questions and the C++ CUED Crib

Exercise 4 - Code Solving (reading files)

Exercise 4 - Code Solving (reading files)
Call `decode_and_print` from your `main` function to shift the characters in the string by 7. You should get the same result as before!

Now change the `main` routine so that it tries decoding with \( N=1 \), then \( N=2 \) up to \( N=25 \), printing the decoded message each time. Do this using a loop. Determine the \( N \) that works for your message.

When you've completed programs 1 to 4 get them marked. Make sure that you have restructured the code - a single 40-line `main` function isn't good enough.

**Exercise 5 - Word lengths**

Your next task is to read the words in a file, find their lengths and print the frequency of word-lengths from 1 to 10. Call the program `wordlengths.cc`

**Tips**

- Create a file with some words in it - just a few words initially, several per line if you want. This file needs to be in the same folder as your program. Remember to save the file.
- Read the words one by one into a string using code like this

```cpp
#include <fstream> // so that file-reading works

string str;
ifstream fileInput; // a variable of a type that lets you input from files
fileInput.open("filename"); // this tries to open a file called "filename"
// The next line uses 'not', a keyword
if(not fileInput.good()) { // print an error message
    cout << "Couldn't open the file."
    cout << endl;
    cout << "Does one exist in the same folder as your program?" << endl;
    return(1);
}
// the code below reads the next word from the file and puts it into the
// variable 'str'. It does so while there are words left in the file
while(fileInput >> str) {
    // print it out to check that the code's working
    cout << str << endl;
}
```

Check that this works as expected before going on to the next stage. You'll need to write a `main` function to contain this code, and `include` some files at the top.

If you want to know how this works in more detail, read a book or ask a demonstrator.

- Create an array of variables called `frequency` to store the frequencies. Easiest is to arrange things so that `frequency[1]` is used to store the number of words of length 1, `frequency[2]` is used to store the number of words of length 2, etc. Initialise these variables to 0. Change your code in the `while` loop so that 1 is added to the appropriate variable each time you've read in a word and found its length. For example, if the length of a word is 3 you'll need to add 1 to `frequency[3]`. More generally, if the word length is `len`, you need to add 1 to `frequency[len]`

Once all the words have been read, this array of variables will hold the final frequency counts.

- Print a simple table of the results like the following, using a loop

```markdown
<table>
<thead>
<tr>
<th>Length</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>
```

- When you have this working with a little file of words, try it with a bigger file - you could even try downloading something from [Project Gutenberg](http://www.gutenberg.org/). What are you going to do if `len` is more than the number of elements in the `frequency` array?

If you haven't got the hang of using arrays to store how many times things happen, read the [I don't understand how count things and store the frequencies in an array](http://www-h.eng.cam.ac.uk/help/tpl/languages/CUED-1A-C++-coursework-20of41-07162014-0121PM).
By putting extra lines at the top of your file you gain access to further functions that have already been written. Here are some examples.

- **#include <cmath>**
  
  With this you can use maths routines - \( \sin(x) \) (which uses radians); \( \text{pow}(x,y) \) (which raises \( x \) to the power \( y \), \( \text{sqrt}(x) \), etc.

- **#include <cstdlib>**
  
  With this you can access functions that generate random numbers. Before the random number generator is used for the first time, call
  
  ```c
  srand(time(0));
  ```

  so that you get a different sequence of random numbers each time you run the program. Don’t call `srand()` more than once. Each time the function `random()` is called, it will return a random positive integer (in the range 0 to 32767 or so). Work out what the following function does and how it works.

  ```c
  int RollDie() {
    int randomNumber, die;
    randomNumber = random();
    die = 1 + (randomNumber % 6);
    return die;
  }
  ```

  You’ll be using this code later, so if you’ve any doubts about what this code does, write a `main` function that calls it, add some `cout` statements to print useful variables out, then build and execute the code.

  The numbers produced by the `random()` routine are only pseudo-random. Here we’re using the computer’s real time clock as the seed. See [wikipedia’s Random number generation page](https://en.wikipedia.org/wiki/Random_number_generation) if you want more details. For more information about functions in general, see

  - [A more sophisticated animation](https://en.wikibooks.org/wiki/C%2B%2B_Programming/Animation) (which might only work in the DPO)
  - [An answer to a frequently asked question](https://en.wikibooks.org/wiki/C%2B%2B_Programming/FAQ)

### Standard data types and mixing types

We’ve already mentioned that there are several types of C++ variables: `int` for integers, `float` for floating point numbers, `char` for characters and `bool` for booleans (`true` or `false`). There are also `doubles` (which are floating point numbers too, but potentially more accurate than `floats`) and `longs` (which store integers that might be too big or small to fit into an `int` variable). You can also declare variables which will only store non-negative values by using the `unsigned` keyword. For example, `unsigned int height;` creates a variable whose contents won’t be negative. `size_t` is a datatype used by many C++ routines when a non-negative integer is being used (e.g. for the length of a string). C++ is quite strict about types. What does the following program print out?

```c
#include <iostream>
using namespace std;

int main() {
  int i=1;
  int j=2;
  cout << "i/j= " << i/j << endl;
}
```

Did you hope it would print out `i/j=0.5`? Actually it prints out `i/j=0` because in C++ an arithmetic operation with only integer operands results in a (possibly rounded-down) integer. If at least one operand is a floating point number, the answer’s a floating point...
number, so you could make this program print out \( i/j=0.5 \) by changing it to

```cpp
#include <iostream>
using namespace std;

int main() {
    int i=1;
    int j=2;
    cout << "i/j= " << i*1.0/j  << endl;
}
```

Computers usually store floating point numbers using a base 2 representation. Just as 1/3 can't be expressed in base 10 using a finite number of digits, so there are many numbers that computers can't accurately store in base 2 using a few bytes. The next exercise illustrates the difficulties.

### Exercise 6 - Accuracy  
[ back to contents ]

In maths, \( x \times 11 - x \times 10 \) will always be \( x \). Will the result be the same on computers?

In a `main` function create a floating point variable called `number` and set it to 0.1. Create a loop that will run 20 times. Inside the loop, reset `number` so that it becomes 11 times itself minus 1, then print out the number and how many times the loop's been executed - e.g.

```
number = 0.1 after 5 iterations
```

Compile and run the program and record the results. Now try 2 changes

- Change `number` so that it's a `double`. Recompile and re-run the program, recording the results.
- Finally, initialise `number` to 0.5 and reset it so that it becomes 11 times itself minus 5. Recompile and re-run.

Try to explain these findings (there'll be more about this issue later).

### Exercise 7 - Pi (math functions)  
[ back to contents ]

You can calculate \( \pi \) just by dropping a pen, as long as you drop it onto floorboards several times. If the pen is as long as the floorboards are wide, then \( \pi \) will be roughly

\[
2.0 \times \frac{\text{number of drops}}{\text{number of times pen lands on crack}}
\]

(look up "Buffon's needle" for the theory). The pen will hit a crack (an edge of a floorboard) if the closest distance (\( D \)) from the pen's centre to a line is less than or equal to 0.5 times \( \sin(\theta) \), where \( \theta \) is the angle between the pen and the cracks.

This gives us the chance to do a simulation (a common task in engineering). Don't try to write the complete program before trying to build it. Take it stage by stage, compiling and testing as you go.

1. In a file called `pi.cc` write a function to simulate the dropping of the pen. It could have the prototype

```cpp
bool dropthepen();
```

returning `true` if it touches a crack in the floor and `false` otherwise.

There are 2 random factors to take account of - the angle of the pen (0-90 degrees) and the distance of the pen's centre from a line (between 0 and 0.5; our floorboards will be 1 unit wide). The function will need to calculate \( \sin(\theta) \). The C++ `sin`
function expects its argument to be in radians. You can get round that by doing

```
float angleindegrees=(90.0*random())/RAND_MAX;  // a number between 0 and 90
float angleinradians=angleindegrees*M_PI/180;  // a number between 0 and pi/2
```

These expressions use variables made available by the inclusion of `cstdlib` and `cmath`, namely RAND_MAX (the biggest integer that the random routine produces), and M_PI (the value of pi).

Create a variable D and set it to \((0.5\cdot \text{random()})/\text{RAND\_MAX}\) (a random number between 0 and 0.5).

Using D and \(\sin(\text{angleinradians})\) you can now work out whether the pen lands on a crack.

2. Add a main routine that calls your function. Before you call the function, call `srandom(time(0))` once to initialise the random number generator (remember, you'll need `#include <cstdlib>` at the top of the file to access these random number routines).

3. Before the while loop, create a variable (numberofHits, say) to store the number of times the pen crosses a crack. Initialise it appropriately. Inside the while loop, add 1 to it when the dropthepen function returns true. After the while loop, write the code to estimate pi. If pi comes out to 3, make sure you're using 2.0 rather than 2 when calculating (when you divide an int by an int in C++, you get an int). By involving a floating point number in the calculations, floating point arithmetic will be performed.

4. Now do 10,000 runs. You should get a more accurate answer for pi.

Your program is likely to have the following layout - some included files, a prototype and 2 functions.

```
#include <iostream>
// other included files

// prototype
bool dropthepen();

// main function
int main {
  // Initialise random number generator, call dropthepen many times,
  // gather statistics and print the answer
}

// dropthepen function
bool dropthepen() {
  // return true if pen lands on crack, otherwise return false
}
```

Exercise 8 - Monopoly © [back to contents]

You're playing Monopoly (if you don't know the game or where the hotels are, read the notes). First the good news - you've landed on GO so you get £200. Now the bad news - there are 5 hotels along the first side of the board and you own none of them. What chance do you have of reaching the 2nd side without landing on any of the hotels? What's the most likely number of hotels that you'll land on?

You could solve this using probability theory. We're going to do it "by experiment", writing a program called `monopoly.cc` to run 10,000 simulations.

Whenever you have a non-trivial program to write, think about how it can be broken down into stages, and how you can check each stage.

1. You've already seen the RollDie function that simulates the rolling of a single die. Copy it into your new file. Now write a function called Roll2Dice to simulate the rolling of 2 dice (call RollDie twice and return the sum of the answers). Before going any further, test it. If it doesn't work, neither will your full program! Here's a main function you could use to test it

```c++
int main() {
```
#include <cstdlib>  

srandom(time(0));  
cout << "Roll2Dice returns " << Roll2Dice() << endl;  
}

You'll need to add prototypes for `RollDie` and `Roll2Dice` too.

2. Write a function to simulate a single user's attempt to get past the hotels. Call it `runTheGauntlet` unless you can think of a better name. It won't need any inputs, but it needs to return the number of hotels landed upon. Think first about your strategy in words, then express that strategy using C++. It helps to have a variable (called `location`, say) that records where you are. First set a counter `hotelsVisited` to 0. You need to keep rolling the dice until you've gone past the 9th square (so use a `while` loop). Each time you move, check to see whether you've landed on a hotel. If you have, add 1 to `hotelsVisited`. At the end of the function return `hotelsVisited`. Test this function - run it from the `main` function a few times and print the outcome to see if the results are reasonable.

To see whether you've landed on a hotel, use something like

```c++
if (location==1 or location==3 or location==6 ...
```

and not

```c++
if (location==1 or 3 or 6 ...
```

The latter isn't illegal but it doesn't do what you might expect.

3. Now call that function 10,000 times using something like

```c++
int output=runTheGauntlet();
```

in a loop. You don't want to print each outcome but you do want to store how many times no hotels were landed on, how many times only 1 hotel was landed on, etc. Create an array called `frequency` to store the results. `frequency[8]` will contain the number of runs when no hotels were landed upon, so for each run when no hotels are landed on you need to add 1 to this value. When you've done all the runs, `frequency[8]` will contain the final value. You need to deal similarly with the other elements of the array. How many elements will this array need? What should the initial value of each element be?

4. Using a loop, print the summary of results on the screen.

```
<table>
<thead>
<tr>
<th>Hotels-visited</th>
<th>Frequency</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

You don't have to print the columns out neatly, but if you want to do so, 2 facilities might help
- `setw` - this lets you set the minimum number of characters produced by the next piece of output.
- `setfill` - with this you can choose the character that will fill the gaps caused by using the `setw` command

So if you want to print out an integer `i` to fill a column 10 characters wide, filling the gaps with blanks, you could use

```c++
#include <iomanip>
...
cout << setfill(' ')  << setw(10) << i ;
```

Note: On each turn in Monopoly, 2 dice are thrown and their sum used to determine how far a player's counter moves. If the GO square is counted as position 0, then the hotels are located on positions 1, 3, 6, 8 and 9 along the first side. In the Cambridge Edition of Monopoly, the properties along the first edge are (in order) "Histon Road", "Parker's Piece", "Hills Road", "Cheddars Lane" and "Bateman Street". As an extra challenge you could find out which property you are most likely to land on, or the consequences of adding the rule that rolling 3 consecutive doubles puts you in jail.

Call by Reference  [ back to contents]
Suppose we want to write a function that will triple the value of a given variable. We could try the following:

```cpp
#include <iostream>
using namespace std;

// prototype
void triple(int i);

int main()
{
  int i=3;
  cout << "In main, i is " << i << endl;
  triple(i);
  cout << "In main, i is now " << i << endl;
}

void triple(int i) {
  i=i*3;
  cout << "in triple, i becomes " << i << endl;
}
```

But it doesn't work as we wanted - the `i` in `main` doesn't change. Run it if you're not convinced. The problem is that `main`'s `i` is a different variable to the `i` in `triple` - each is a variable that's local to the function it's in. That the 2 variables have the same name is a coincidence (if the `i` variable in the `triple` function were renamed, the code would work just as well). The only link between the 2 variables is that when `triple` is called, `triple`'s `i` gets its initial value from `main`'s `i` - `i` is being passed "by value".

If we want to change `main`'s `i` we need to "call by reference" - note the added ampersands in the following code.

```cpp
#include <iostream>
using namespace std;

// prototype
void triple(int& i);  // added ampersand

int main()
{
  int i=3;
  cout << "In main, i is " << i << endl;
  triple(i);  // NO added ampersand
  cout << "In main, i is now " << i << endl;
}

void triple(int& i) { // added ampersand
  i=i*3; // here i is an alias for main's i
}
```

When a variable is passed using "call by value", the function is given the variable's value. The function doesn't know what the original variable was, so the function can't change it. When a variable is passed using "call by reference" the function can "refer to" the original variable, so it can be changed.

Why is "call by reference" useful? One reason is that it lets functions "return" more than one value. If, for example, you write a function that has 3 input parameters that are passed "by reference", the function can change all 3 of them, and those changes will be visible outside the function.

### Alternative notations

C++ has alternative ways to do some things:

- **Comments** - Instead of using `//` to comment out a line, you can use `/* ... */` to comment out a block of text.
- **Increment/decrement** - There are shortcuts to changing a variable's value.
### More Loops

In a `while` loop you sometimes want to abort early. There are 2 commands to do this:

- **break** - this breaks out of the loop completely
- **continue** - this breaks out of the current cycle and starts the next one.

The best way to understand these commands is to see them in action. If you run this program, what would it print out? If you’re not sure, **run it and see**!

```cpp
#include <iostream>
using namespace std;

int main() {
    int i=0;
    while(i<10) {
        i=i+1;
        if(i==2)
            continue;
        if (i==4)
            break;
        cout << "i=" << i << endl;
    }
    cout << "End of looping" << endl;
}
```

Another way to do looping is to use a `for` loop. Earlier we had this `while` loop.

```cpp
int num=1;
while (num<11) {
    cout << num << endl;
}```
Notice that it has

- Initialisation code - `int num=1`
- Code to control termination - `num<11`
- Code that's run each cycle to make the next cycle different - `num=num+1`

With a `for` loop all this code that controls the cycling is brought together in a compact form. Here's the `for` loop equivalent of the above `while` loop

```cpp
for (int num=1; num<11; num=num+1) {
    cout << num << endl;
}
```

or more commonly

```cpp
for (int num=1; num<11; num++) {
    cout << num << endl;
}
```

Note the format -

```cpp
for (initialisation ; decision ; each cycle )
```

"for" loops are more common than `while` loops (people like having the "loop-controlling" code together) so you'll have to get used to them. Try the "for" loop teaching aid to get some practise and try re-writing some of the earlier exercises using `for` loops instead of `while` loops.

C++'s `for` loop is very flexible. The "loop variable" doesn't need to start at 1, nor do you need to add 1 to it each time you go round the loop. For example, the following code prints 11, 10 ... 1.

```cpp
for (int num=11; num>0; num--) {
    cout << num << endl;
}
```

[hide extra information]

Note that the `num` variable in the code above is created within the `for` loop, and no longer exists when the `for` loop ends; it's local to the loop.

More about Arrays  

An array can be passed to a function as an input parameter. Arrays are always "passed by reference". Here's an example.

```cpp
#include <iostream>
using namespace std;

void timesarrayby2(int numbers[]); // The square brackets are needed
   // because 'numbers' is an array. Note that there's no ampersand
int main() {
    int nums[10];
    for(int i=0; i<10; i++) {
        nums[i]=i;
    }
    timesarrayby2(nums);
    // Note that the numbers in the array have changed.
}
```
for(int i=0; i<10; i++) {
    cout << nums[i] << endl;
}

void timesarrayby2(int numbers[]) {
    for(int i=0; i<10; i++) {
        numbers[i]=2*numbers[i];
    }
}

Earlier we used 1-dimensional arrays but you can create arrays with more dimensions. Here's an example of a 2D array that stores 6 integers in 2 rows of 3 columns. If you want to set all of the elements to 0 you could use “nested loops” - loops inside loops. The following code sets table[0][0] to zero, then table[0][1] to zero, etc

for (int row=0; row<2; row++)
    for(int column=0; column<3; column++)
        table[row][column]=0;

More about Strings

strings are quite sophisticated variables. You've already seen how if you have a string called s you can find its length by calling the s.length() function, but there are many other string functions too. The fragments below illustrate a few of them

string s="a few words";
// starting at position 7, extract a substring of s that is 2 characters long,
string t=s.substr(7,2); // t is now "or" because positions are counted from 0

int found=s.find('w'); // find the position of the first w

// The next line uses a special constant "string::npos" - see
// the documentation about strings if you really need to know more
if (found==string::npos) {
    cout << "There is no w in the string" << endl;
} else {
    cout << "There's a w in position " << found << endl;
}

found=s.find_last_of('w'); // look for the last w
if (found==string::npos) {
    cout << "There is no w in the string" << endl;
} else {
    cout << "The last w is in position " << found << endl;
}

More Decisions

There are older, common alternatives to the boolean operators

<table>
<thead>
<tr>
<th>This</th>
<th>is equivalent to</th>
<th>this</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>...</td>
<td>and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
You can "nest" if statements. In the following code, the 2nd if line is only reached if `num < 5` is true.

```cpp
if (num < 5) {
    cout << "num is less than 5" << endl;
    if(num < 3) {
        cout << "and num is less than 3" << endl;
    } else {
        cout << "but num is equal to or greater than 3" << endl;
    }
} else {
    cout << "num is greater than or equal to 5" << endl;
}
```

Follow the route that the execution of this code takes when `num` is 6, then 4, then 2.

- Sometimes you might want to perform a different action for each of many possible values of an integer variable. You could use `if` statements a lot of times. Alternatively you can use `switch`. Here's an example:

```cpp
#include <iostream>
using namespace std;

int main() {
    int i=0;
    while(i<10) {
        switch (i) {
        case 0: cout << "i is zero" ;
            break;
        case 1: cout << "i is one" ;
            break;
        case 2: cout << "i is two" ;
            break;
        default: cout << "i isn't 0, 1, or 2";
            break;
        }
        i=i+1;
        cout << endl;
    }
}
```

When `i` is 0, the "case 0" block is run. In this situation the `break` doesn't break out of the surrounding `while` loop, it breaks out of the `switch`. Without the break, execution would "fall through" into the case 1 block. Try to work out what happens in this code before running it.

- If you decide to quit from the program completely, use `exit(1);` (in the `main` function, `return` will do this, but it's a special case. For this to work you need `#include <cstdlib>` at the top of your file.

### Variable scope and lifetime

The scope of a variable is the region of code from which a variable can be accessed. Variables created in a function or a loop generally cease to exist when the loop or function ends. Furthermore, variables inside one function can't be accessed from another function. This feature helps you write safer code. If you create a variable at the top of a file outside of all functions, it will be available to all of your functions all the time. These so-called `global` variables are best avoided.

### Classes

If the range of types that C++ provides is too restrictive you can invent new types. When you use a language (like English or C++) you
are often attempting to represent or model the real world. The more the modelling language structurally resembles what it's modelling, the easier the modelling is likely to be. If for example your program was dealing with 2-dimensional points, it would help to have a type of variable to represent a point. In C++ you can create such a type like this

```cpp
class Point {
    public:
        float x;
        float y;
};
```

Note that this doesn't create a variable, it creates a new type of variable. Point is called a class. To create a variable of type Point you do the same kind of thing that you did when creating variables of type int, etc. To create an int called i you would do

```cpp
int i;
```

To create a Point called p you do

```cpp
Point p;
```

The following code fragment shows how to set p's component fields to values.

```cpp
p.x=5;
p.y=7;
```

Whereas in arrays all the elements needed to be of the same type and each element was identified by a number, in classes they can be of different types and are each given a name. Suppose you have the following data

<table>
<thead>
<tr>
<th>Name</th>
<th>Anna</th>
<th>Ben</th>
<th>Charlie</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height</td>
<td>1.77</td>
<td>1.85</td>
<td>1.70</td>
</tr>
<tr>
<td>Age</td>
<td>20</td>
<td>18</td>
<td>15</td>
</tr>
</tbody>
</table>

You can create a class designed to contain this information as follows

```cpp
class Person {
    public:
        string name;
        float height;
        int age;
};
```

You could then create a variable to represent Anna's information by doing

```cpp
Person anna;
anna.name="Anna";
anna.height=1.77;
anna.age=20;
```

If you wanted to print Anna's height later on, you could do
cout << "Anna's height = " << anna.height << endl;

Note that the following doesn't work because cout doesn't know what to do when asked to print a non-standard thing like a Person

cout << anna << endl;

You need to print each component individually. If we have several people, it's useful to create an array of Persons. The syntax when creating arrays of new types is the same as when creating arrays of built-in types like ints. The following line creates an array called family big enough to store information about 3 people:

Person family[3];

To set the age of the first person to 20 you'd do

family[0].age = 20;

Classes are heavily used in C++. The components of the classes above are public (i.e. visible from outside the object) but they needn't be. Classes can also contain functions. You can also create new classes from old by using inheritance. And you can make "cout << anna << endl;" work with some extra code. But all that's for the future.

Exercise 9 - A phone directory

This program revises file-handling and gets you to create and use your own datatypes. Create a text file at least 10 lines long where each line contains an integer (a phone number) and a name, with the name in double-quotes. For example, the file might begin

332746 "Tim Love"
999 "Emergency Services"

- Design a data structure using class to contain a number and a name (how many elements should the structure contain? What types should they be? You can store the number in a string or an int - the latter leads to more work).
- Create an array of these data structures. If you've called the array directory the beginning of the array would look rather like the diagram. The array needs to be big enough to store all the entries.
- Write some code that reads the information from the file and stores it in the appropriate place in the array. Make sure that you don't try to store more lines of data than you have space for in your array.
- At the end, after all the information has been stored, visually check that the information that's finally in the array matches the data in the file by printing out the information that is in the array.

Tackle the task a step at a time, checking your work as you reach each milestone. You've already used the getline function to read a line of text from a file into a string variable. getline returns true only if it
You may sometimes want to access the individual bits of a byte. You'll need to do so when programming robots in the 2nd year, and questions about bits are often in 1st year exams. A byte contains 8 bits, each of which can be off or on (a binary digit - 0 or 1). The program below shows how to display the bits of a byte. It uses the & (aka bitand) operator, which performs a bit-wise and with the 2 operands (there's also a | (aka bitor) operator, which performs a bit-wise or).

```cpp
#include <iostream>
using namespace std;

int main() {
    unsigned char num=43;
    unsigned char bitmask=128; // the bit pattern 10000000
    for (int thebit=7; thebit>-1; thebit--) {
        // Now see if (num bitand bitmask) is non-zero
        if (num bitand bitmask) {
            cout << "1";
        } else {
            cout << "0";
        }
        bitmask=bitmask/2;
    }
    cout <<endl;
}
```

It goes through the bits of the byte called `num` checking the value of each bit (most significant first) and printing it out, so that in the end you get a binary representation of the decimal number 43.
You can use the bit operators to switch bits off or on. If you do \texttt{num bitand x} and \(x\) is all 1s, then the answer will be the same as \texttt{num}. If \(x\) has exactly 1 bit set to 0, then \texttt{num bitand x} will be the value of \texttt{num} with that bit set to 0. So if you wanted to set the 4th bit from the right to 0 in \texttt{num} you could do

\[
\texttt{num} = \texttt{num bitand 247;}
\]

Similarly, to set the 3rd bit from the right to 1 you could do

\[
\texttt{num} = \texttt{num bitor 4;}
\]

As you can see, for natural numbers there's an obvious way to store the value - as binary. It's not so obvious how to deal with negative integers - see Wikipedia's Two's complement page for details. And there's always the problem that \texttt{int} values are usually stored in 4 bytes, so there's a limit how accurately you can store integers. There's sometimes a \texttt{long long} type available, which uses 8 bytes but that still won't give you unlimited range.

A \texttt{float} variable occupies 4 bytes too. It's not obvious how those bytes might be used to store reals, and even more so than with \texttt{int} there are accuracy issues. The code below (which goes beyond what you need to know about C++) displays the bits of a floating point number byte by byte. You can use it to study floating point representation (note that on PCs the most significant byte is at the highest memory address - i.e. it's printed out last by this program)

```cpp
#include <iostream>
using namespace std;

void printbinary(unsigned char num) {
    unsigned char bitmask=128;
    for (int thebit=7;thebit>-1;thebit--) {
        if (num bitand bitmask)
            cout << "1";
        else
            cout << "0";
        bitmask=bitmask/2;
    }
}

int main() {
    unsigned char* cp;
    float f=43;
    cp=reinterpret_cast<unsigned char*>(&f);
    for (int byte=0;byte<4;byte++) {
        cout << "Byte " << byte << " = ";
        printbinary(*(cp+byte));
        cout << endl;
    }
}
```

If you run this code it will print out

```
Byte 0=00000000
Byte 1=00000000
Byte 2=00101100
Byte 3=01000010
```

Those values in base 10 are 0, 0, 44, 66. How do those relate to the real number 43? It's all described in the IEEE 754 specification. If
you try the IEEE 754 Converter you'll get the idea. One bit denotes the sign, 8 bits represent the exponent (using 2 as the base) and the other bits represent the mantissa (the value). You're not expected to memorize this format, but you should be aware of the consequences of it; namely that there are limits to the range and accuracy of the numbers. Just as 1/3 can't be represented in base 10, so 0.1 can't be precisely represented in base 2. So be careful when you use real numbers! See Floating-point Basics for more information.

The formula is
\[
\text{sign} \times 2^{\text{exponent}} \times \text{mantissa}
\]

The exponent here is the stored 8-bit value minus 127. The mantissa used is 1 plus the 23-bit stored value (whose most significant bit represents 0.5). There are special bit-patterns to represent special values like infinity.

If you're really keen read What Every Computer Scientist Should Know About Floating-Point Arithmetic.

Enumerations

Enumerations are another way to make code easier to read. Earlier we had an array of integers created using `int month[12];`. If we wanted to set the July value to 31 we could do

```cpp
month[6]=31;
```

(remember, element indexing begins at zero). Alternatively we could use enumerations.

```cpp
enum Month {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
month[Jul]=31;
```

`Month` is a new variable type whose only legal values are `Jan, Feb ... Dec`. These values are just aliases for 0, 1 ... 11, but they're easier for humans to read.

Writing to files

To be able to use the file-writing facilities you need to add

```cpp
#include <fstream>
```

to the top of your file. To illustrate writing to files, we'll write the values of the `month` array (one value per line) into a file called `myData`. Notice that the line that writes to the file is similar to the use of the `cout` command when writing on the screen - you just replace `cout` by the name of the `ofstream` (which in this example is `fileOut`).

```cpp
ofstream fileOut;  // create a variable called 'fileOut' to store info about a file.
                   // It's not an int or a float, it's an ofstream; a special type
                   // of variable used when writing to files
fileOut.open("myData"); // try to open the file for writing
if (not fileOut.good()) {
    // if there's a problem when opening the file, print a message
    cout << "Error trying to open the file" << endl;
    return;
}
// The computer will only reach here if the file's ready to use, so use it
for(int i=0; i<12; i++) {
```
fileOut << month[i] << endl; //write to the file
}
fileOut.close();

The **good()** function of an **ofstream** variable returns **false** if the last file operation failed.

**Semicolons**  

By now (and with the help of the compiler's error messages) you've probably gained a lot of experience about where semi-colons are needed. Note that

```
if (i==3)
```

doesn't need a semi-colon after it because it's not a completed statement. Unfortunately, though it doesn't require a semi-colon, it's not illegal to have one. The following is legal C++

```
int i=0;
if (i==3) {
    cout << "i is 3" << endl;
}
```

and will print out **i is 3** even though **i** is 0. Why? Well, the **if(i==3)** code is an incomplete construction. It expects a statement after it, and in this context a semi-colon is a null-statement. The following layout shows more clearly what happens.

```
int i=0;
if(i==3)
;
{
    cout << "i is 3" << endl;
}
```

The **cout** line is run every time because it's not being controlled by the **if**. There's a similar risk with **while**. The compiler won't complain that the following code runs forever

```
int i=0;
while (i<10); {
    i++;        }
```

The trouble is that the **i++;** line isn't inside the body of the loop, so **i** is always 0. The following layout shows more clearly what happens

```
int i=0;
while (i<10)
;
{
    i++;        }
```

See the **I'm confused about commas and semi-colons** frequently-asked-question for details.

**Exercise 10 - Bubble sort**  

This exercise introduces no new programming concepts - it's practice at using loops and comparisons, converting ideas expressed in
English into a program written in C++.

Using the ideas from the talk (or the web - try Wikipedia), write a program that uses the bubble sort algorithm to sort the phone directory created in the previous exercise (you may as well start this exercise by copying the code from exercise 9 - you need only add about 10 lines to it to finish this exercise). Sort by the phone numbers, smallest first (or if you've stored the numbers as strings, sort them in alphabetical order - you can compare strings alphabetically using < or >). Keep comparing the numbers in neighbouring entries, swapping the complete entries if they're in the wrong order. Remember to stop scanning the values once they're all in the right order. While developing the program it might help to print the values in the list after each pass, to see if the correct entries are being swapped. How many passes are needed in the worst case?

You'll need to be careful when swapping entries. Consider this little program that tries to swap integers (you'll be swapping phone directory entries, but the same problem arises).

```cpp
int main() {
    int a =1, b=2;
    // Let's try to swap a and b
    a=b;
    b=a;
}
```

This doesn't work. If you dry-run through it you'll find that both a and b have the value 2 at the end. You'll need to create an extra, temporary variable.

You can add the extra code to your main or create a separate function. If you have created an entry class to contain a person's information, and you created an array of entries called directory then the following might be a reasonable prototype if you want the put the bubblesort code into a function.

```cpp
void bubblesort(entry directory[], int num_of_entries)
```

The first input parameter has the square brackets to show it's an array. You might call the function by doing bubblesort(directory, 5);

Note that though you need to compare the "number" fields of the entries, you can swap complete entries - you needn't separately swap the number and name fields.

Exercise 11 - Binary search

Another revision exercise. Don't start this until you're sure that bubblesort works. You could start by copying the code from exercise 10.

Write a program that asks the user to type in a number. If that number is in the phone directory you've created earlier, the program should print the corresponding name. If the number isn't in the directory, the program should print a suitable message.

Use the binary search method (so you'll need to sort the items first). Look at the value of the middle item in the list, compare that to the value you're seeking and decide which half of the list you're going to repeat the process in. You might find it useful to create variables (low and high for example) to store where in the array the ends of the current section are, updating one of them each time you reduce the section. Keep going until you've found the item or you can't search any further. While you're developing the program it might be useful to print out low and high each time you change them. Test the function, making sure it does something sensible when asked to look for a number that doesn't exist.

Algorithmic Complexity

So far your programs have taken minutes to write and milliseconds to run, so optimising wasn't crucial. Before long however, you'll be writing programs where efficiency matters. For functions like searching and sorting that process lots of data there are vast differences between the efficiency of the algorithms - pick the wrong one and your program will run for days rather than minutes. It's important to determine how the time taken depends on the amount of data, n. If the
association's linear, then doubling the amount of data will double the program time. Sadly however, many of the methods that are easy to write have times proportional to the square of the data size. Even if your bubble sort can sort 10 numbers quickly, sorting a 1,000 will take 10,000 times longer.

One way to express the efficiency of an algorithm (its Algorithmic Complexity) is to use "Big-O" notation. An \( O(1) \) algorithm's time doesn't depend on the amount of data, an \( O(n) \) algorithm scales linearly, an \( O(n^2) \) algorithm quadratically, and so on. "Big-O" notation indicates the order of magnitude of the performance, so constants aren't put in the brackets. It usually describes worst-case behaviour.

More precisely, if the number of actual steps the program takes to run when the input is size \( n \) is denoted by \( f(n) \), then we say that \( f(n) \) is \( O(n) \) if there is a constant \( c \) and input size \( N \) such that \( f(n) < c \cdot n \) for all \( n > N \). More generally, any function can be inside the \( O() \), so we can write \( f(n) \) is \( O(g(n)) \) if \( f(n) < c \cdot g(n) \) for all \( n > N \). Commonly used are \( O(n^k) \) for some small constant \( k \), \( O(\log(n)) \), \( O(\exp(n)) \) and combinations of these.

The efficiency of an algorithm can be deduced by studying the code. That's not always trivial, but here's a simple example.

```cpp
// Initialise a square matrix called b of size n
for (int i=0; i < n; i++) {
    for (int j=0; j < n; j++) {
        b[i][j] = 99;
    }
}
```

By considering how many times each loop is executed (note that one loop is nested inside the other) you should be able to deduce how many times the assignment statement is run and hence the order of this task. Then look at your binary search code (or just think about the algorithm) and work out its order (hint - it's not \( O(n) \) like a naive search would be. Think about the effect of doubling the amount of data).

**Exercise 12 - Measuring program speed**

The C++ standard library has a built-in function to sort items, which often uses the quicksort algorithm. We'll investigate how the time taken to sort items depends on \( n \), the number of items. Is the time proportional to \( n \) or \( n^2 \)? The claim for the built-in function is that it's \( O(n \log(n)) \) (natural logs, but it doesn't really matter). We'll determine by experiment whether this is true, then compare that to the speed of your bubble sort code.

The program below uses some commands you've not seen before, but they'll be useful to you next term and/or next year. Like next term, we're providing you with a program that works but is incomplete. We're also not going to provide step-by-step instructions. The new features involve

- **Timing** - We've provided a routine you can use to time parts of your programs. It returns the number of microseconds that have elapsed since 1st Jan 1970. Don't worry about how it works.
- **vector** - C++'s vector is a more sophisticated version of an array. You'll be using it in the 2nd year. We're using it here because it makes the code simpler. You won't need to change any of the lines that use vector in this code. Just note that you can read values from a vector just as you'd read them from an array, using square brackets.
- **Standard functions** - C++ provides functions to sort, search, shuffle, etc, so it's worth seeing how to use them rather than having to write your own functions all the time.

Here's a program to sort 10 numbers

```cpp
#include <iostream>
#include <vector>
#include <sys/time.h>
#include <cmath>      // needed for log
#include <iomanip>    // needed for setw
#include <algorithm>  // needed for sort
```
using namespace std;

// Prototypes
long microseconds();
long timeTheSort(long thelength);

int main() {
    long thelength=10; // the number of items: n in the description above
    long timeTaken;
    timeTaken=timeTheSort(thelength);

    // Output the results, formatting so that each number is in
    // a column 10 characters wide.
    // First print the column headings
    cout << " Length     Time     Time/n     Time/(n*log(n))" << endl;
    cout << setfill(' '); // fill the gaps with spaces
    cout << setw(10) << thelength << endl;
}

long timeTheSort(long thelength) {
    // Create a vector called numbers big enough to store 'thelength' ints
    // You can access the values in it just as if 'numbers' were an array
    vector<int> numbers(thelength);
    long thetime; // needed later

    // After the next lines, the vector number will contain
    // the integers 0 to thelength-1
    for (int i=0; i<thelength; i++) {
        numbers[i]=i;
    }

    // Randomize and output the values
    random_shuffle(numbers.begin(), numbers.end());
    for (int i=0; i<thelength; i++) {
        cout << numbers[i] << endl;
    }

    // Sort and output the values
    sort(numbers.begin(), numbers.end());
    for (int i=0; i<thelength; i++) {
        cout << numbers[i] << endl;
    }
    return thetime;
}

// This function returns microseconds elapsed since the dawn of computing
// (1st, Jan, 1970). Don't worry about how it works
long microseconds() {
    struct timeval tv;
    gettimeofday(&tv,0);
    return 1000000*tv.tv_sec + tv.tv_usec;
}

Compile and run this code. You'll see a list of unsorted numbers, a list of sorted numbers, then part of a table. Make sure that you understand the flow of the code before proceeding. Note that microseconds() isn't used yet - you'll need it later. If you don't know what long means, re-read the Standard data types section.

First, comment out the lines that print the numbers (later you won't want to watch 10,000,000 numbers being printed out). Then
complete the line of the table by calling the `timeTheSort` function. You'll need to add lines to the `timeTheSort` function to call `microseconds()` twice - once just before calling `sort` and once just after - and find the difference between the 2 results, returning the answer. Time only the `sort` function - don't time the setting-up code too. It should take less than 10 ms. Then complete the final 2 columns of the table. If `Time/N` comes out to 0, you need to read the Standard data types section again. Your output should be something like

<table>
<thead>
<tr>
<th>Length</th>
<th>Time</th>
<th>Time/n</th>
<th>Time/(n*log(n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8</td>
<td>0.8</td>
<td>0.347436</td>
</tr>
</tbody>
</table>

Using an appropriately placed `for` loop add a completed line of data to the table for lengths of 100, 1000, etc up to 10,000,000.

Which length has the smallest time-per-element value? Is the time proportional to `n*log(n)`?

Now assess the performance of bubblesort. Take the lines from Exercise 10 that performed the sorting, and use them here instead of C++'s `sort` routine. You'll need to adjust your bubblesort code so that it sorts an appropriately-sized array of numbers rather than an array of directory entries. You needn't create a new function, but if you really want to, a reasonable prototype for it would be

```cpp
void bubblesort(vector<int> numbers, int thelength);
```

Instead of working out `Time/(n*log(n))`, calculate `Time/(n*n)` (because the speed of bubble sort is supposed to be proportional to `n^2`). You'll find that bubblesort is much slower than `sort`, so don't sort an array any bigger than 10,000 items. Run the program and look at the results. Is the performance of bubblesort close to the predictions? Can you think of any reason for ever using bubblesort?

Finally, change the code so that the table is written into a file called "ex12table". We'll be checking to see that you've produced the file, so don't delete it afterwards.

**[hide extra information]**

As an extra challenge, try measuring the speed of sorting your phone directory entries. You have 2 main issues to resolve

- You'll need to create a long list of entries. You can adapt the given code to create random telephone numbers.
- The `sort` routine can only work if it knows how to compare the items it's sorting. This isn't a problem when the items are a standard type, but if they're a user-defined type, the user needs to provide a user-defined comparison routine. More precisely, the `<` operator needs to be defined for the user-defined type. This is possible to do in C++, though the notation isn't trivial. Here's an example

```cpp
bool operator <(const Phonenumbers& entry1,const Phonenumbers& entry2 ) {
    return entry1.number < entry2.number;
}
```

**More exercises**

The early exercises here could be done during the course to re-enforce your understanding of basic concepts. The later exercises might be useful after the course to help you prepare for next term.

**Simple**

- Print the odd integers between 0 and 100 in ascending order
- Print the odd integers between 0 and 100 in descending order
- Get the user to type in a word. Print the word with the letters reversed
- Get the user to type in a word. Count the number of vowels in the word
- Rewrite all the earlier exercises that used `while` loops so that they use `for` loops instead.

**Classes**

- Invent a class called `fraction` suitable for storing a vulgar fraction. Write a function that given a variable of type `fraction` prints out the value in the form `a/b`. Write a function that given 2 variable of type `fraction` prints their sum.

**Functions**
**Puzzle-solving**

- By trial and error, find all the Pythagorean triples where the integers are less than 100 \((a,b,c)\) is a Pythagorean triple if \(a^2 + b^2 = c^2\). It's easy to eliminate duplicates \((4, 3, 5)\) is a duplicate of \((3, 4, 5)\). It's rather harder to eliminate multiples (e.g. \((6, 8, 10)\) is a multiple of \((3, 4, 5)\), but the whole program should be less than 20 lines long.
- Pick a number. If it's even, divide by 2. If it's odd multiply by 3 and add 1. Continue this until you reach 1. E.g. if you start with 3, the sequence will be 3-10-5-16-8-4-2-1. Write a program to find out which integer less than 100 produces the longest chain.
- Ask the user to type an integer in the range 1 to 7. If the user types something invalid, ask them to try again. Then print the day of the week corresponding to the number - print "Sunday" if the user types 1, etc.
- The zeta function, \(\zeta(s)\), can be evaluated by summing terms of the form \(1/i^s\) for \(i\) running from 1 to infinity. It can also be evaluated by multiplying terms of the form \((1/(1-1/p))\) where \(p\) runs through all the primes. See how true this is for series of 10 terms then 100 terms.
- Simulate a situation where you keep asking different people their birthdays until you find 2 people with the same birthday. Assume all years have 365 days. Make the code into a function so that you can run it many times and find an average for the number of people you need to ask.
- Big numbers are often printed with commas after every 3 digits, counting from the right - e.g. 1,345,197. Write a function which is given a float and prints out the number with commas. You'll need to find out how to convert numbers to strings - use the WWW!
- Create an array to represent an 8x8 chessboard. In chess a Bishop can move as far as it likes along diagonals. Write a program to count the number of places a Bishop can move to for a particular starting position. From which starting positions does it have the greatest choice of moves?
- In chess, a knight moves in an 'L' shape (2 squares along a row or column, then one square left or right). Write a program to count the number of places a Knight can move to for a particular starting position.
- Starting at any square on a chessboard, move the knight randomly, and keep moving until it lands on a square it's already visited. How many moves does it do on average?
- Write a function that adds 2 vulgar fractions and prints their sum as a vulgar fraction. E.g. its prototype could be

```cpp
void addfractions(int numerator1, int denominator1, int numerator2, int denominator2);
```

so that calling it as `addfractions(3,4,5,6)` will make it output something like \(38/24\), or better still \(19/12\), or even \(1 + 7/12\). You could create a class called Fraction.

- There are many variants of "Poker Dice" - Yahtzee™ for example. The player rolls 5 dice, keeps as many as they want, rolls the remaining ones, keeps as many of those as they want, then rolls again. The aim is to get particular combinations of die-values. To keep things simple we're just going to collect 6s. The chance of throwing 5 6s in the first roll is \(1/6^5\). Using the rolls as described above, what's the chance of having 5 6s at the end of the 3 rolls? We'll determine the answer experimentally, running 100,000 attempts.

Here is the code for most of the main routine.

```cpp
int successes=0;
for (int i=0;i<100000; i=i+1) {
    if (fiveSixesThrown())
        successes=successes+1;
}
cout << "After 3 rolls, the success rate is " << successes/1000.0 << "%";
```

From this, work out the prototype of the `fiveSixesThrown` function, then write the function (use the `RollDie` function that you've already used). Think about the variables that the function might need - I used `totalNumberOfSixes, numberOfDiceLeft, and thisRollNumberOfSixes`, but it's up to you.

Then complete the program.
Rather than 3 rolls, how many rolls will give you a 50% chance of ending up with 5 6s? The tidiest way to determine this is to change the code so that your `fiveSixesThrown` function is given an argument to show how many rolls are allowed. You shouldn't need to make many modifications to the function's code. Then you can write code in `main` like the following to increase the number of rolls until you reach the required level of performance:

```c
float percentage=0;
numOfRolls=1;
while (percentage<50) {
    if (fiveSixesThrown(numOfRolls)) {
        ...
    }
    numOfRolls=numOfRolls+1;
}
```

Write a program that simulates the rolling of 10 dice 10000 times. Display a frequency table of outcomes (the outcome being the sum of the 10 dice) - e.g.

```
Outcome  Frequency
1       0
2       0
...
60      1
```

(this was the final exercise in the Mich 2008 term)

Useful Links

- CUED's C++ page
- CUED's IA Computing Help
- CUED Tutorial Guide to C++ Programming
- CUED's C++ Frequently Asked Questions
- cppreference.com's C++ Language Tutorial
- Introduction to C++ (MIT OpenCourseWare) - see especially the Assignments
- Installing free compilers