Stream Basics

- Streams are a way of reading data from a device or sending it to a device
  - Later we will use streams to read and write data from files
- cout is the standard out, which is usually the screen, but can be redirected to other outputs, such a file or a printer
- cin is the standard in, which is usually the keyboard, but can be redirected from a file for instance
- cerr is the “error” stream, which is not buffered (cout are buffered)

Stream Basics

- You can send data to a stream using << and receive data from a stream using >>
- cout can only receive data (i.e. only << can be used with it) and cin can only send data (i.e. only >> can be used with it)
  - Later we will encounter file streams that can both send and receive data
- endl is used to indicate a new line. A new cout does not cause a new line in the output

Special Characters

- There are some special characters that can be used in strings (must be inside the string "..."):
- \n: New line (essentially the same as endl)
- \t: Tab space
- \": Displays a " character
- \': Displays a ' character
- \\: Displays a \ character

Comments

- In any programming it is useful to add comments to your code so that you (and other people) can know what you did and why
- You can either comment out sections using /* … */ where everything between /* and */ is ignored by the compiler
- … or you can use // ,where compiler ignores the rest of the line that occurs after //

Time to write some code!

- Name your C++ file “read_two_stdin.cpp”

Pointers

- Pointers are something that Python doesn't have at all
- A pointer is a memory address
- Pointers are declared by putting a star in front of the variable name.
  - E.g. a pointer to a memory location that contains an integer would be declared as follows: `int *P1;`
- The pointer (i.e. memory location) of any variable can be obtained using the & symbol
  - E.g. If `A` is a simple variable, then `&A` is the pointer to the memory location where `A` is stored
- Conversely, if you have a pointer, the `*` symbol can be used to access the value stored at that memory location
  - E.g. If `P1` is a pointer to an integer, then `*P1` is the integer stored at the memory location `P1`

An example with simple pointers

```
#include <iostream>

using namespace std;

int A,*P1,*P2,B;

void main(void)
{
    A=200;

    P1=&A;

    P2=&B;

    *P2=*P1;

    cout<<"B: "<<B<<endl;
}
```

Sets P1 equal to the memory location where A is stored

Sets P2 equal to the memory location where B is stored. That B hasn't been given a value is immaterial as declaring it assigns memory to the variable.

Sets the value stored at memory location P2 equal to the value stored at memory location where P1. This has the effect of setting B equal to A.

Pointers and Memory Allocation

- The previous example might seem a bit esoteric and not very useful
  - It will become very important later when we look at advanced memory structure such as linked lists and trees
- In the meantime there is a very useful thing that can be done using pointers:
- Pointers can have memory directly allocated to them, rather than simply pointing at an existing variable
  - This means that we can effectively have arrays where you don’t need to know the size of the array when writing the program, but can set its size based on user input and/or calculated values

Allocating Memory

- Memory allocation is done using the new command
  - It can also be done using malloc, though this is considered to be more of a C, rather than C++ method
- Any memory that is allocated should be de-allocated using delete
- To allocate memory for a single variable:

E.g. To allocate memory for a single integer:

```
int *P;
P=new int;
*P=24;
delete P;
```

- Memory can also be allocated to a pointer to be used like an array:

E.g. For a pointer containing enough memory to store 20 integers:

```
int *P;
P=new int[20];
P[12]=24;
delete[] P;
```

Arrays and Pointers

- Arrays are closely related to pointers and can be though of as being a pointer with pre-allocated memory (there are subtle differences, but not enough to worry about):
  - Internally they are treated slightly differently as arrays are allocated within the stack (a smallish piece of memory automatically assigned to a program by the operating system), while dynamically allocated memory is allocated within the heap
  - Dynamically allocated variables can thus be used to allocate essentially any memory (including virtual memory) available to the operating system, while the total size of static arrays that you can use in a program is far more limit
    - In Windows the default stack size is 1MB (though it can be changed)
- The name of an array variable is also a pointer to the memory location of the first item in the array
  - i.e. If A is an array (e.g. int A[20]), then A≡&A[0]

Pointers to Pointers

- Sometimes you might want to use a pointer to a pointer (i.e. the memory location where a memory location's value is stored)
  - As a pointer is indicated by a * during its declaration, a pointer to a pointer is indicated by **
  - E.g. To define a pointer to a pointer to a memory location containing a double:

```
double **P
```

Dynamic Allocation of Higher Dimensional Arrays

- A common use for a pointer to a pointer is to use it as a 2-dimensional array.
- This requires that memory be assigned for an array of pointers
  - Note the type casting that is required as new returns a pointer rather than a pointer to a pointer. E.g if the array is to by max_i by max_j in size (with both these variables type int):

```
P=(double **) new double*[max_i];
```

- Each of entries in the array of pointers now needs memory to be allocated to it
  - Note that by allocating memory in this way means each column need not be the same length, though in this example they are

```
for (i=0;i<max_i;i++) P[i]=new double[max_j];
```

Passing Pointers to Functions

- Functions can have pointers as parameters
  - This can be used be used to get the function to return more than one value, but in C++ this is done more easily by passing a variable by reference (see next section)
    - By giving a pointer to a variable, the function can write a value to the memory location indicated by the pointer and thus return an extra value.
  - This is also a very useful way for a function to take in an array (remembering the equivalence between arrays and pointers)

Time to write some code!

- Name your C++ file “array_2D.cpp”

Streams and Data Files

- We have used two streams already, namely cin and cout
- We can create our own streams to input and output data
- Streams are variables (strictly speaking they are objects, though more on that later in the course)
  - For files the variable type is fstream (need to include the header <fstream>)
- Stream variables are declared like any other. E.g. To create a stream called myFile:

```
fstream myFile
```

Opening Data Files

- Simply creating a stream variable is not enough, we need to associate it with a file, which must either be created or opened
- This is done using open:

e.g. myFile.open("test.txt",fstream::out);

  - As stated before, fstream is not a simple variable, but actually a class of which open is a member function (or method), but much more on this later when we look at classes in C++ in detail
- The first parameter in open is the name of the file to be opened
- The second parameter is a set of flags saying how the

Opening Files

- There are a number of flags that can be set when opening a file stream:

fstream::in: Open file as an input stream

fstream::out: Open file as an out stream

fstream::binary: Open the file for binary output (it is set to text output by default). **More on this later**.

fstream::app: Append the output to the end of the file

fstream::trunc: Clear the file on opening (this is done by default if out is set, but not if in or in and out are set)

- These flags can be combined using the | symbol (| is a bitwise or). E.g. To open a file for reading and writing and to

Checking and Closing Files

- After open has been called it is good practice to check if the file was opened successfully
  - A file might not open if, for instance, it is already open in another program or if you are trying to open for reading a file that does not exist
- After open has been called, the member function fail will return true if the file couldn't be opened

- After a file stream is finished with it should be closed using the close member function. This does 2 things:

Time to write some code!

- Open a text file in notepad and input the data:

“1,2,3,4,5,6,16,18,21,17,15,12”

- Save it as “interp_data.txt”
- Name your C++ file “read_file.cpp”

Functions by Reference

Passing values to functions by reference

- In a normal function the value in the argument is copied into parameter and they are 2 separate variables (note again that this is a code fragment)
- This program will return "2 4 800" as the value stored in, for instance, a is simply a copy of that stored in var1 and thus they are two completely separate variables, with a change in the value of a having no influence on the value var1

```
int f_no_ref(int a, int b)
{
	a=a*10;
	b=b*10;
	return(a*b);
}


void main(void)
{
	int var1=2, var2=4, ans;
	ans=f_no_ref(var1,var2);


	cout<<var1<<" "<<var2 <<" "<<ans<<endl;
```

Passing values to functions by reference

- Placing a & symbol either in front of the variable or just after the variable type (i.e. either int& a or int &a is fine):
- This time the program will return "20 40 800" as values stored in, for instance, a are no longer a copy of that in var1, but is a reference to it. This means that a is now effectively another name for var1
- Any change to the argument will thus change the value stored in the parameter variable

```
int f_ref(int &a, int &b)
{
	a=a*10;
	b=b*10;
	return(a*b);
}
void main(void)
{
	int var1=2, var2=4, ans;
	ans=f_ref(var1,var2);


	cout<<var1<<" "<<var2 <<" "<<ans<<endl;
```

Passing values to functions by reference

- Why pass values by reference?
  - A function can return more than 1 value
  - Passing by reference can be slightly quicker (the value does not need to be copied to a new memory location)
- Why not always pass values by reference?
  - Sometimes you don't want to have the value of your argument change (or, if someone else is going to use the function, even the potential for it to change)
  - You can't give an expression as an argument for a function where the value is passed by reference:

e.g. Using the previous code fragments:

`ans=f_no_ref(var1*3+10,var2/7);` is fine as it is only the value that is being passed to the function

`ans=f_ref(var1*3+10,var2/7);` won't work as, for instance, `var1*3+10` can't be evaluated to give a variable who's value can be changed (though, `ans=f_ref(var1=var1*3+10,var2=var2/7);` will work as the expressions are being evaluated to a variable before they are passed to the function)

Do Sheet 2 Exercise 3

Interpolation

- In ACS3 you learnt a bit about interpolation!
- Lagrange and chebyshev interpolation
- Write some C++ code to implement these algorithms
- Use the data from interp_data.txt for the lagrange, and create your own data for the chebyshev
- Use the code from read_data.cpp to read it in
- Assume the first 6 values are x values, the last 6 are y

Interpolation

- Completing this is also your homework
- Write two versions of your interpolation functions
- One of which accepts a single interpolation point as input (along with anything else you require) and returns the value of the interpolating polynomial at that point as a return value
- The other version accepts a number of interpolation points and returns the values of the polynomial at those points, passed by reference

Interpolation

- Verify your method works correctly by passing in the original x_values and check the interpolating polynomial returns exactly the the y_values. Compute the 2-norm of the difference between your values
- ADVANCED – Read https://stackoverflow.com/questions/776283/what-does-the-restrict-keyword-mean-in-c
- How might the (non standard) restrict keyword impact the performance of your function if you were doing the verification above?

Interpolation

- VERY ADVANCED – Try using the restrict keyword and see if it gives you performance differences when using a large amount of data to interpolate/data points to interpolate onto. Consider giving different optimisation flags to the compiler. Use the timing methods we used in array_2D.cpp
- SUPER ADVANCED – Examine the assembly in your executable and try and determine if there are differences which might explain your performance

Interpolation

- VERY ADVANCED – Try using the restrict keyword and see if it gives you performance differences when using a large amount of data to interpolate/data points to interpolate onto. Consider giving different optimisation flags to the compiler. Use the timing methods we used in array_2D.cpp
- SUPER ADVANCED – Examine the assembly in your executable and try and determine if there are differences which might explain your performance