Using C++ classes

C++ classes

- The types `double` and `int` etc. are too restrictive. What about complex numbers, strings, matrices . . . ?
- `#include <complex>` to use complex numbers. Actually, we won't need this.
- `#include <string>` to use strings.
- `#include <sstream>` to use strings efficiently.
- `#include <vector>` to work with vectors.
- `#include <fstream>` to work with files.
- Matrices? Sorry, you have to write your own!

You can write your own custom types. That is the main thing C++ programmer's actually do.
Add all these `#include` statements to `stdafx.h`. We can then use all of these libraries easily. We'll assume `using namespace std;` throughout.

Using a vector - slide 1

```
// create a vector
vector<double> myVector;

// add three elements to the end
myVector.push_back( 12.0 );
myVector.push_back( 13.0 );
myVector.push_back( 14.0 );

// read the first, second and third elements
cout << myVector[0] <<"\n";
cout << myVector[1] <<"\n";
cout << myVector[2] <<"\n";
```

Remember C++ programmer's count from 0. This is one reason why.

Using a vector - slide 2

```
// change the values of a vector
myVector[0] = 0.1;
myVector[1] = 0.2;
myVector[2] = 0.3;

// loop through a vector
int n = myVector.size();
for (int i=0; i<n; i++) {
   cout << myVector[i] <<"\n";
}
```

Note we start counting from zero.

Using a vector - slide 3

```
// Create a vector of length 10
// consisting entirely of 3.0's
vector<double> ten3s(10, 3.0 );

// Create a vector which is a copy of another
vector<double> copy( ten3s );
ASSERT( ten3s.size() == copy.size());

// replace it with myVector
copy = myVector;
ASSERT( myVector.size() == copy.size());
```

Passing big objects around

When you write a function that takes a vector parameter you should write it like this:

```
double sum( const vector<double>& v ) {
    double total = 0.0;
    int n = v.size();
    for (int i=0; i<n; i++) {
        total += v[i];
    }
    return total;
}
```

- It would be a good idea to learn this program off by heart.
- Notice the strange const and & symbol. We need these because vectors are too big to keep copying all the time.

Pass by value

```
void printNextValue( int x ) {
    x = x + 1;
    cout << "B: Value of x is "<<x<<"\n";
}

void main() {
    int x = 10;
    cout << "A: Value of x is "<<x<<"\n";
    printNextValue( x );
    cout << "C: Value of x is "<<x<<"\n";
    return 0;
}
```

Pass by reference

```
void printNextValue2( int& x ) {
    x = x + 1;
    cout << "B: Value of x is "<<x<<"\n";
}

void main() {
    int x = 10;
    cout << "A: Value of x is "<<x<<"\n";
    printNextValue2( x );
    cout << "C: Value of x is "<<x<<"\n";
    return 0;
}
```

- For very small data types (`double`, `int`, `bool`), pass by value is quicker.
- For everything else, pass by reference is quicker.
- But there is a danger of confusing code.

Pass by const reference

```
void printNextValue( const int& x ) {
    x = x + 1;
    cout << "B: Value of x is "<<x<<"\n";
}
```

This code does not compile

Another use of pass by reference

C++ does not allow you to return multiple values. You can use pass by reference to get round this.

```
void polarToCartesian( double r, double theta,
                       double& x, double& y ) {
    x = r*cos(theta);
    y = r*sin(theta);
}
```

```
static void testPolarToCartesian() {
    double r = 2.0;
    double theta = PI/2;
    double x=0.0,y=0.0;
    polarToCartesian(r,theta,x,y);
    ASSERT_APPROX_EQUAL( x,0.0,0.001 );
    ASSERT_APPROX_EQUAL( y,2.0,0.001 );
}
```

Writing to a file

```
// create an ofstream
ofstream out;

// choose where to write
out.open("myfile.txt");

out << "The first line\n";
out << "The second line\n";
out << "The third line\n";

// always close when you are finished
out.close();
```

Works just like `std::cout` except for the open and closing.

Passing a stream as a parameter

Pass a reference to an ostream.

```
void writeHaiku( ostream& out ) {
    out << "The wren\n";
    out << "Earns his living\n";
    out << "Noiselessly.\n";
}

void testWriteHaiku() {
    // write a Haiku to cout
    writeHaiku( cout );
    // write a Haiku to a file
    ofstream out;
    out.open("haiku.txt");
    writeHaiku( out );
    out.close();
}
```

An ofstream is an ostream.

Working with strings

```
// Create a string
string s("Some text.");

// Write it to a stream
cout << s<< "\n";
cout << "Contains "
     << s.size() <<
     " characters \n";
// Change it
s.insert( 5, "more ");
cout << s <<"\n";

// Append to it with +
s += " Yet more text.";
cout << s <<"\n";

// Test equality
ASSERT( s=="Some more text. Yet more text.");
```

Technical points about strings

- When you write text in double quotation marks you obtain data of type `char*`. This means a pointer to a memory address containing a sequence of characters.
- We'll cover pointers in detail later in the course.
- C++ will automatically cast this to a `string` under most circumstances.
- Using a `string` is better than using a `char*` because they're more efficient and have lots of helpful functions.
- Use `\"` to write quotation marks inside quotation marks. Use `\\` to write backslashes inside quotation marks.

Working with strings efficiently

Using + to build up strings is slow. Don't do this:

```
string s("");
for (int i=0; i<100; i++) {
    s+="blah ";
}
cout << s<<"\n";
```

Do this:

```
stringstream ss;
for (int i=0; i<100; i++) {
    ss<<"blah ";
}
string s1 =ss.str();
cout << s1 <<"\n";
```

A stringstream is an ostream.

Writing a chart in C++

Solution 1: Just write the data and create the chart in Excel.

```
void writeCSVChartData( ostream& out,
        const vector<double>& x,
        const vector<double>& y ) {
    ASSERT( x.size()==y.size());
    int n = x.size();
    for (int i=0; i<n; i++) {
        out << x[i] <<","<<y[i] <<"\n";
    }
}
void writeCSVChart( const string& filename,
        const vector<double>& x,
        const vector<double>& y ) {
    ofstream out;
    out.open( filename.c_str() );
    writeCSVChartData( out, x, y );
    out.close();
}
```

Using classes in header files

To make this part of a library we need to declare it in the header.

```
void writeCSVChart( const std::string& filename,
        const std::vector<double>& x,
        const std::vector<double>& y );
```

Unfortunately, you should never write `using namespace std;` in a header file so all these `std::` prefixes are required. Boring!

Writing a chart in C++

Solution 2: Create a web page containing a chart.

- Create a file called `myPieChart.html`. Open it with a text editor (e.g. Notepad)
- Visit `https://google-developers.appspot.com/chart/interactive/docs/quick_start`.
- Copy the code example into your file.
- Save the file.
- Open the file in a web browser.

What does this file do?

- I am not going to tell you in detail!
- We're learning C++ from the bottom up. Let's learn web development the easy way.
- Guess how to change the chart to display what you want.
- See if you were correct.

Writing a charting function steps

1) Create a header file `charts.h`.
2) Create a C++ source file called `charts.cpp`.
3) Add placeholders for testing.
4) Write functions to write the charting boiler plate.
5) Write a simple version of the interesting bit of code.
6) Test the pie chart works in a browser.
7) Write a final version of the interesting bit of code.
8) Write a test for the interesting code.
9) Write a function that wraps it all together.
10) Add that function to your header file.

Step 1 - the header file

What are the required steps when writing a header file?

Step 1 - the header file

- Right-click on "header files" to create.
- Call the file `charts.h`.
- All header files should start with `#pragma once`.
- Include standard libraries with `#include "stdafx.h"`.
- (We'll cover tests later.)

```
#pragma once

#include "stdafx.h"
```

Step 2 - the C++ source file

What are the required steps when writing a source file?

Step 2 - the C++ source file

- Right click on "source files" to create.
- Call the file `charts.cpp`.
- All source files should `#include` the header.
- (We'll cover tests later.)

```
#include "charts.h"
```

Step 3 - add placeholders for testing

When creating new files, how do you build in testing? This will depend upon your testing framework, of course.

Step 3 - add placholders for testing

In charts.h:

```
void testCharts();
```

In main.cpp

```
int main() {
    testMatlib();
    testGeometry();
    testCharts();
    testUsageExamples();
}
```

In charts.cpp:

```
void testCharts() {
}
```

Step 4 - an easy functions

- We pass an `ostream&` reference to the function.
- We use `\"` to escape quotes in quotes.
- The spacing in HTML files isn't very important, so this function doesn't reproduce the spacing of Google's example pie chart precisely.

```
static void writeTopBoilerPlateOfPieChart( ostream& out ) {
    out<<"<html>\n";
    out<<"<head>\n";
    out<<"<!--Load the AJAX API-->\n";
    out<<"<script type=\"text/javascript\"";
    out<<"src=\"https://www.google.com/jsapi\">";
    out<<"</script>\n";
    out<<"<script type=\"text/javascript\">\n";
    out<<"google.load('visualization', '1.0',";
    out<<" {'packages':['corechart']});\n";
    out<<"google.setOnLoadCallback(drawChart);\n";
    out<<"function drawChart() {\n";
    out<<"var data=new google.visualization.DataTable();";
    out<<"\n";
    out<<"data.addColumn('string', 'Label');\n";
    out<<"data.addColumn('number', 'Value');\n";
}
```

Step 5 - easy versions of the remaining code

- Writing a function for the bottom boiler plate code is just as easy.
- Writing a function `writeFixedPieChartData` that prints out the data for a fixed pie chart is easy too. The harder bit will be making it work with changing data.
- Let's "cheat" for now, and write this easy function so we can see if we can write a chart to file that works in a browser.
- This is a sensible practice. Work in small pieces. Once you've solved one simple problem, move on to the next simple problem.

Step 5 - The simplified solution

```
static void writeFixedPieChartData( ostream& out) {
    out<<"data.addRows([\n";
    out<<"['Bananas', 100],\n";
    out<<"['Apples', 200],\n";
    out<<"['Kumquats', 150]\n";
    out<<"]);\n";
}
```

Step 6 - Writing a test

```
static void testFixedPieChart() {
    ofstream out;
    out.open("FixedPieChart.html");
    writeTopBoilerPlateOfPieChart(out);
    writeFixedPieChartData( out );
    writeBottomBoilerPlateOfPieChart( out );
    out.close();
}
```

```
void testCharts() {
    TEST( testFixedPieChart );
}
```

- We've written enough code to test. So, let's run it.
- If you run this test in Visual Studio it will create the file in the same folder as `main.cpp`.

Step 7 - The interesting code

Given a string of labels produce output that looks like this:

```
data.addRows([
['Bananas', 100],
['Apples', 200],
['Kumquats', 150]
]);
```

- We'll assume the labels don't contain quotation marks or other special characters.
- Note that the last line is special - there is no comma.

Step 7 - Write the interesting code

```
static void writeDataOfPieChart( ostream& out,
                                 const vector<string>& labels,
                                 const vector<double>& values) {
    out<< "data.addRows([\n";
    int nLabels = labels.size();
    for (int i=0; i<nLabels; i++) {
        string label = labels[i];
        double value = values[i];
        out<<"['"<<label<<"', "<<value<<"]";
        if (i!=nLabels-1) {
            out<<",";
        }
        out<<"\n";
    }
    out<<"]);\n";
}
```

Step 8 - Testing the interesting code

How can we test the interesting bit of code?

Step 8 - Test page 1

We first create a string containing the actual data.

```
static void testPieChartData() {
    // this test automates the checking
    stringstream out;
    vector<string> labels(3);
    vector<double> vals(3);
    for (int i=0; i<3; i++) {
        stringstream ss;
        ss<<"A Label "<<i;
        labels[i] =ss.str();
        INFO( labels[i] );
        vals[i]=(double)i;
    }
    writeDataOfPieChart( out,
              labels,
              vals );
    string asString = out.str();
```

Step 8 - Test page 2

We then compare it against a string containing the expected data.

```
    stringstream expected;
    expected<<"data.addRows([\n";
    expected<<"['A Label 0', 0],\n";
    expected<<"['A Label 1', 1],\n";
    expected<<"['A Label 2', 2]\n";
    expected<<"]);\n";
    string expectedStr = expected.str();
    ASSERT( asString==expectedStr );
}
```

- Since `fstream` and `stringstream` are both types of stream, the interesting code was easy to test.
- It is perfectly possible to write meaningful tests for almost any code. If you can't test it, you've designed your code incorrectly or don't understand the problem properly.

Step 9 - Write a function that wraps it all together

This is essentially the same code as the function `testFixedPieChart`:

```
void pieChart( const string& file,
               const vector<string>& labels,
               const vector<double>& values ) {
    ofstream out;
    out.open(file.c_str());
    writeTopBoilerPlateOfPieChart(out);
    writeDataOfPieChart( out, labels, values );
    writeBottomBoilerPlateOfPieChart( out );
    out.close();
}
```

Step 10 - Add the function to the header file

```
void pieChart( std::string& file,
               std::vector<std::string>& labels,
               std::vector<double>& values );
```

This is copied from the code in the .cpp file except we've had to put in lots of `std::` statements since you should never write `using namespace std;` in a header file.

Software Architecture

- We created our chart by writing a web browser file rather than writing our own graphics code.
- This idea is behind the entire design of the World Wide Web!
- Servers receive text data (from users filling in forms and typing URLs).
- Servers produce text data (HTML files).
- Its all very easy to debug and test, because it all happens through text files.
- You could easily adapt our library to be used in a web app.

Use C++ for what C++ is good at (e.g., fast calculations), but use other languages where appropriate (e.g., user interfaces, prototyping).

Summary

By putting everything we've learned together, we can write something very sophisticated.

- Use `vector<double>` for a vector. Pass them as `const vector<double>&`.
- Use `string` to represent strings. Pass them as `const string&`.
- Use `stringstream` to build complex strings.
- Use `fstream` to write to files.
- A `stringstream` an `fstream` and `cout` are all examples of `ostream`. Pass them as `ostream&`.
- Sometimes you might want to drop the `const` when passing vectors and strings, but not often.
- Don't return by reference (yet...).