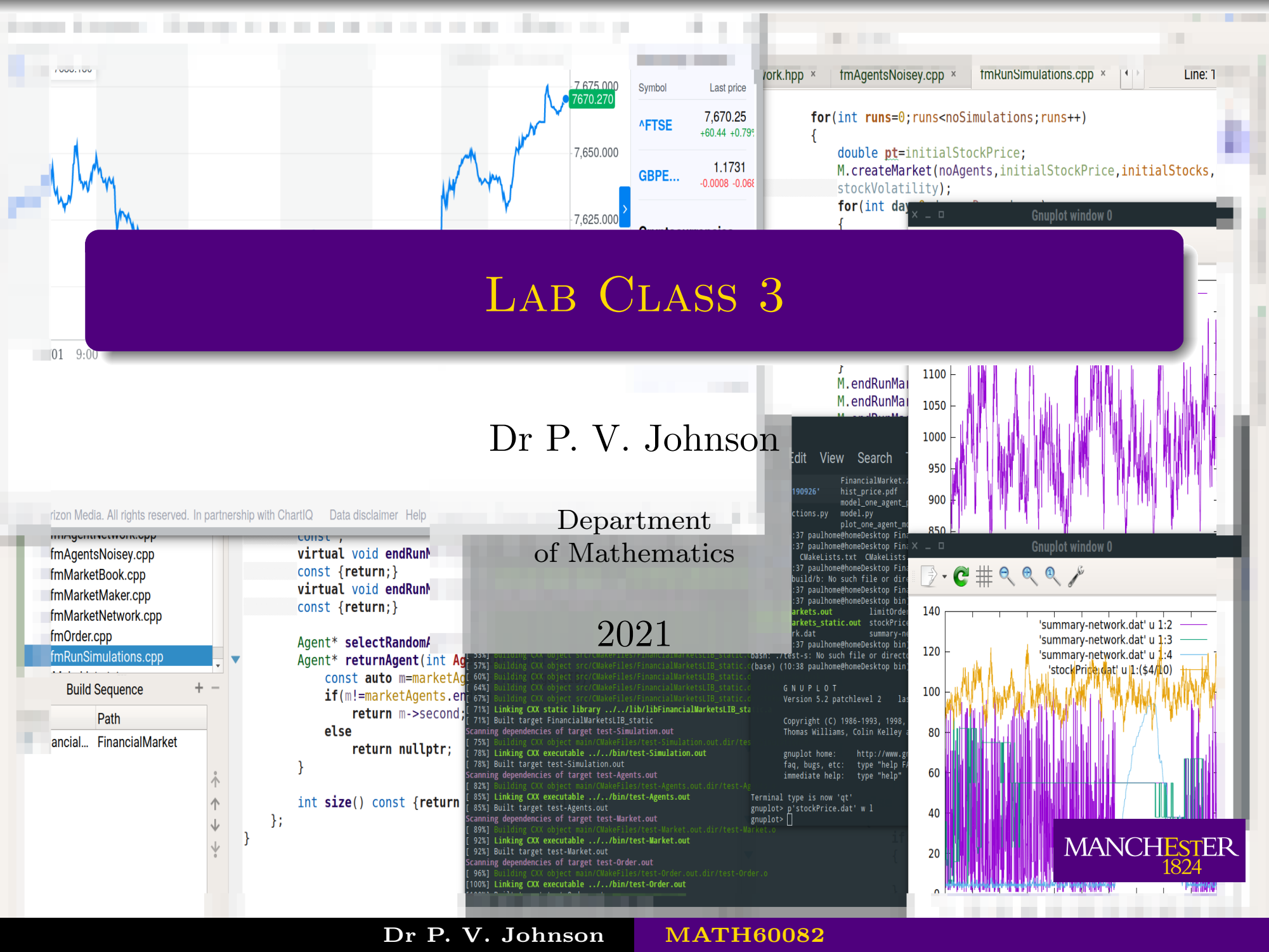
# Lab Class 3

Dr P. V. Johnson

Department
of Mathematics

2021



# Last Time

- How to output results to a file
- Example 2.1 Solving an ODE numerically
- Example 2.2 How to verify the accuracy of a solution
- Example 2.3 Discussion - Mini Task 2

MANCHESTER 1824

# This Week

- Random numbers
- Monte Carlo
- European Options

MANCHESTER 1824

- A computer can **only** generate a random sequence of integers, but of course we can then take that sequence of integers and convert it to any required distribution.
- Some of the conversions are simple but others are more complex, luckily we now have inbuilt c++ conversion to all standard distributions (more on this later).

# Random Numbers

- A computer can **only** generate a random sequence of integers, but of course we can then take that sequence of integers and convert it to any required distribution.
- Some of the conversions are simple but others are more complex, luckily we now have inbuilt c++ conversion to all standard distributions (more on this later).
- This means you will always have to create a generator to pass as an argument to the probability distribution you want to generate.

MANCHESTER 1824

# Random Numbers

- Create a new project and include the `random` library, the `cmath` library for any calculations and also `iostream` to show results onscreen.

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

int main()
{
  // declare a random number generator
  // here we choose the merton twister engine (32bit)
  mt19937 rng;
  cout << rng() << endl;
  // output is 3499211612
}
```

download

# Random Numbers

- Create a new project and include the `random` library, the `cmath` library for any calculations and also `iostream` to show results onscreen.
- Then declare a variable of type `mt19937`.

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

int main()
{
  // declare a random number generator
  // here we choose the merton twister engine (32bit)
  mt19937 rng;
  cout << rng() << endl;
  // output is 3499211612
}
```

download

# Random Numbers

- Create a new project and include the `random` library, the `cmath` library for any calculations and also `iostream` to show results onscreen.
- Then declare a variable of type `mt19937`.
- This declares a new random number generator, which generates pseudo random sequence of integers defined by the Mersenne Twister algorithm.

```
#include <iostream>
#include <cmath>
#include <random>
using namespace std;

int main()
{
  // declare a random number generator
  // here we choose the merton twister engine (32bit)
  mt19937 rng;
  cout << rng() << endl;
  // output is 3499211612
}
```

download

# Random Numbers

- Create a new project and include the `random` library, the `cmath` library for any calculations and also `iostream` to show results onscreen.
- Then declare a variable of type `mt19937`.
- Once you have declared the generator, the next number in the sequence is called with the `operator()` function, or as can be seen in the program to the right.

```
#include <iostream>
#include <cmath>
#include <random>
using namespace std;

int main()
{
  // declare a random number generator
  // here we choose the merton twister engine (32bit)
  mt19937 rng;
  cout << rng() << endl;
  // output is 3499211612
}
```

download

- When we call this function the first time the value is 3499211612.

```
cout << rng() << endl;
cout << rng() << endl;
cout << rng() << endl;
// output is
//    3499211612
//    581869302
//    3890346734
```

download

- When we call this function the first time the value is 3499211612.
- You will notice that the number is the same every time the program is run.

```
cout << rng() << endl;
cout << rng() << endl;
cout << rng() << endl;
// output is
//    3499211612
//    581869302
//    3890346734
```

download

# Random Numbers

- When we call this function the first time the value is 3499211612.
- You will notice that the number is the same every time the program is run.
- This is because a random number generator always follows a predefined sequence, and the default start point in the sequence is always the same.

```
cout << rng() << endl;
cout << rng() << endl;
cout << rng() << endl;
// output is
//    3499211612
//    581869302
//    3890346734
```

download

# Random Numbers

- When we call this function the first time the value is 3499211612.
- You will notice that the number is the same every time the program is run.
- This is because a random number generator always follows a predefined sequence, and the default start point in the sequence is always the same.
- We can call the next few numbers by writing a loop or by simple copy/pasting the `cout` line a few times

```
cout << rng() << endl;
cout << rng() << endl;
cout << rng() << endl;
// output is
//    3499211612
//    581869302
//    3890346734
```

download

- This repetition in the sequence is in fact extremely useful when bug checking your code as you can eliminate the randomness as the reason your results change.
- When doing numerical computations it is hardly ever required to start the sequence of with a random start point, since you are only ever interested in what happens on average over a large number of simulations.
- If you wish to change the starting point of sequence to get a different result or to rerun calculations on the same set of random numbers, then we need to set a seed.

# The Seed

- You set the seed with the function `seed(int)`, or see code to the right

```
rng.seed(123);
cout << rng() << endl;
```

download

# The Seed

- You set the seed with the function `seed(int)`, or see code to the right
- This seed does not mean the 123rd number in the sequence,

```
rng.seed(123);
cout << rng() << endl;
```

download

# The Seed

- You set the seed with the function `seed(int)`, or see code to the right
- This seed does not mean the 123rd number in the sequence,
- so seeding with 124 will not give you the next number.

```
rng.seed(123);
cout << rng() << endl;
```

download

# The Seed

- You set the seed with the function `seed(int)`, or see code to the right
- This seed does not mean the 123rd number in the sequence,
- so seeding with 124 will not give you the next number.
- Seeds will normally only need to be set **once** (and only once) in your programs, unless you want to compare results in some way.

```
rng.seed(123);
cout << rng() << endl;
```

download

# The Seed

- There maybe times where you want to set a really random sequence (to generate demonstrations, different results etc) then the best way is to use the type `random_device`.

```
random_device randomSeed; // this uses
    generate a random seed
rng.seed(randomSeed()); // set the seed
cout << rng() << endl;
// my output this time is 1776944864
// but it's different for each run!!!
```

download

# The Seed

- There maybe times where you want to set a really random sequence (to generate demonstrations, different results etc) then the best way is to use the type `random_device`.
- Declare a variable of that type and you can set any number of random seeds with the `operator()` function, see code to the right

```
random_device randomSeed; // this uses
    generate a random seed
rng.seed(randomSeed()); // set the seed
cout << rng() << endl;
// my output this time is 1776944864
// but it's different for each run!!!
```

download

# TASKS

1. Use a loop to output the first 1000 numbers in the sequence.
2. Try resetting the seed using the number 123 inside the loop – what happens?
3. Try resetting the seed using a `random_device` inside the loop – what happens?

MANCHESTER 1824

# Uniform distributions

- From here on in we will be using the default seed which should be consistent across platforms.
- We can now create distributions that be sampled.
- For the uniform distribution we use the syntax as shown in this program

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

int main()
{
  // declare a random number generator
  // here we choose the merton twister engine (32bit)
  mt19937 rng;
  // a uniform distribution
  uniform_real_distribution<double> U(0.,1.);
  // get a number from the uniform distribution
  cout << U(rng) << endl;
  return 0;
}
```

download

# Uniform distributions

- Assume $u \sim U(0,1)$ is a random draw from the uniform distribution, calculate

$$\text{Prob}(0.25 < u < 0.5)$$

- To do this with simulations we need to count how many times $u$ lands in the interval $[0.25, 0.5]$, and then divide by the total simulations.

# Uniform distributions

- Assume $u \sim U(0,1)$ is a random draw from the uniform distribution, calculate

$$\text{Prob}(0.25 < u < 0.5)$$

- To do this with simulations we need to count how many times $u$ lands in the interval $[0.25, 0.5]$, and then divide by the total simulations.
- Your code might look a bit like this

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

int main()
{
  // declare a random number generator
  // here we choose the merton twister engine (32bit)
  mt19937 rng;
  // a uniform distribution
  uniform_real_distribution<double> U(0.,1.);

  // total number of simulations
  int N=1000;
  // keep track of payoff
  double sum=0;
  for(int i=0;i<N;i++)
  {
    double u=U(rng); // generate the random number
    if(0.25 < u && u < 0.5) // split the interval condition
    {
      sum += 1;// add in payoff
    }
  }

  cout << " P(0.25 < u < 0.5) = " << sum/N << endl;
  return 0;
}
```

download

# TASKS

1. Run the code above and check the result makes sense.
2. Put the algorithm into a function like this:

```
double calcProb(double a,double b,int N)
{
 // calculate probability a<u<b
}
```

3. Return the calculated probability for different values of $N$, what can you say about the results as $N \to \infty$?

MANCHESTER 1824

# Normal distributions

- Now try a normal distribution
- Declare the type: `normal_distribution<double>` to enable us to convert random integers into draws from a normal distribution.

# Normal distributions

- Now try a normal distribution
- Declare the type: `normal_distribution<double>` to enable us to convert random integers into draws from a normal distribution.
- The code should look like:

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

int main()
{
  // declare a random number generator
  // here we choose the merton twister engine (32bit)
  mt19937 rng;
  // a normal distribution with mean 0 and variance 1
  normal_distribution<double> Phi(0.,1.);
  // get a number from the normal distribution
  cout << Phi(rng) << endl;
  return 0;
}
```

download

- Now check the code by creating a histogram plot of the normal distribution over the range of intervals

$$a = -4.2 \leq (i - 1/2)h < x < (i + 1/2)h \leq b = 4.2$$

  with $i = -10, -9, \ldots, 9, 10$ and $h = 0.4$.
- Here we have 21 intervals, so we are going to have to store the central position of each of the intervals as well as keep count of the frequency at which a random draw lands in the interval.
- For simplicity we shall use arrays for storage, with a double for the interval position and integer for counting.

# Normal distributions

- Start by declaring some variables to store the values, noting here that $n$ must be constant to use with arrays
- You should then initialise the values.
- Because array indices work starting from 0 it will be better if we index the interval by $j$=0,1,...,20.

```
// input parameters
int totalRuns=1e6; // total number of simulations
// interval specification
const int n=21;  // number of intervals
double a=-4.2,b=4.2;// start/end points
double h=(b-a)/n;// fixed interval width

double x[n];// center of intervals

// local storage
int counter[n];// number of times landing in interval
```

download

Then the centre of the interval is given by the formula

$$x_j = a + \frac{h}{2} + jh \text{ for } j = 0, 1, \ldots, 20 \tag{1}$$

It is also good practice to reset the value of the counter to zero.

```
for(int j=0;j<n;j++)
{
  x[j] = a+h/2.+j*h;   // setup the position of the central points
  counter[j]=0; // reset the counter
}
```

- Next we want to generate a random number and then find which interval it sits in,
- and to do this we need the `floor` function from the math library.
- We can rearrange (1) to find that a number `phi` sits in the $j^*$th interval given by

$$j^* = \left\lfloor \frac{\phi - a}{h} \right\rfloor$$

- We can test this formula with some code:-

```
  double phi=Phi(rng); // get a number from the normal distribution
  int jStar = floor( (phi-a)/h ); // use the floor function to get j* for the
      nearest point to x_j* to phi
  cout  << phi << " " << jStar << " " << a+jStar*h << " " << a+(jStar+1)*h <<
      endl;
// output is
//   0.13453 10 -0.2 0.2
```

download

- We can test this formula with some code:-

```
  double phi=Phi(rng); // get a number from the normal distribution
  int jStar = floor( (phi-a)/h ); // use the floor function to get j* for the
      nearest point to x_j* to phi
  cout  << phi << " " << jStar << " " << a+jStar*h << " " << a+(jStar+1)*h <<
      endl;
// output is
//   0.13453 10 -0.2 0.2
```

download

- The interval selected is the 10th interval which corresponds to [-0.2,0.2], and clearly `phi=0.13453` is in this interval.
- When you are coding this up you will need to check that you are within the bounds.

# TASKS

1. Create a loop in your program to generate n draws from $\phi \sim N(0, 1)$.
2. Use the interval selection code to find which *bin* each random number lands in (`jStar`)
3. increment (or add one) to the appropriate *bin* (`counter[jStar]`)
4. Plot out your results of `x` vs `counter` – does it look normal?

# European Options

- Now we are going to value an European call option using Monte-Carlo. The setup is very simple,
- we just need to sum up the payoffs from a bunch of sample paths and then take the average.

# European Options

- Now we are going to value an European call option using Monte-Carlo. The setup is very simple,
- we just need to sum up the payoffs from a bunch of sample paths and then take the average.
- First start with an empty program except for the random number generator, as follows

```
#include <iostream>
#include <random>
#include <cmath>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  // declare the random number generator
  mt19937 rng;



}
```

download

- Include the `random` library, as well as `cmath` for access to mathematical functions required later.

- Include the `random` library, as well as `cmath` for access to mathematical functions required later.
- Next declare the normal distribution

```
// declare the distribution
normal_distribution<> ND(0.,1.);
```

download

- Include the `random` library, as well as `cmath` for access to mathematical functions required later.
- Next declare the normal distribution
- Declare your parameters for the option at the top of your main function alongside the number of simulations we are going to run (a counting number)

```
// declare the distribution
normal_distribution<> ND(0.,1.);
```

download

```
// parameters
double stock0=9.576,strikePrice=10.,interestRate=0.05,sigma=0.4,maturity
    =0.75;
// number of paths
int N=1000;
```

download

# European Options

- Now we can write the Monte-Carlo algorithm in less than 10 lines
- Initialise a sum variable with zero and then run through each path,
- adding in the payoff of a call for each particular stock price at maturity

```
// initialise sum to zero
double sum=0.;
// run through each simulation
for(int i=0;i<N;i++)
{
  double phi=ND(rng); // get a random draw from N(0,1)
  double ST=S0 * exp( (interestRate - 0.5*sigma*sigma)*maturity + phi*sigma
     *sqrt(maturity) ); // get a random draw for the value of stock price
     at maturity
  sum = sum + max( ST - strikePrice , 0. ); // add in the payoff of the
     option in that case
}
cout << sum/N*exp(-interestRate*maturity) << endl; // output the average
   value over all paths
```

download

# European Options

- Now we can write the Monte-Carlo algorithm in less than 10 lines
- Initialise a sum variable with zero and then run through each path,
- adding in the payoff of a call for each particular stock price at maturity
- Running this code should return a value of 1.2671, which we can check is reasonably close to the analytic value we would expect.

```
// initialise sum to zero
double sum=0.;
// run through each simulation
for(int i=0;i<N;i++)
{
  double phi=ND(rng); // get a random draw from N(0,1)
  double ST=S0 * exp( (interestRate - 0.5*sigma*sigma)*maturity + phi*sigma
     *sqrt(maturity) ); // get a random draw for the value of stock price
     at maturity
  sum = sum + max( ST - strikePrice , 0. ); // add in the payoff of the
     option in that case
}
cout << sum/N*exp(-interestRate*maturity) << endl; // output the average
   value over all paths
```

download

# TASKS

1. Put the Monte Carlo code into a function, and check the result. Use this code if you get stuck:-
   (Click here to download)
2. Run this function several times – what happens?
3. Write the keyword `static` in front of the declaration of the random number generator like this:-
   `static mt19937 rng;`
   and try again. What happens now?
4. Run this code for $N = 100, 200, 300, \ldots$ and plot out the results for different $N$. What does it look like?

MANCHESTER 1824

- Ok now we are confident that we can generate random samples of the solution for a given `N`, we want to see what happens expected distribution of the results as we increase the number of paths.
- We are going to do some analysis now which requires the stl storage container `vector` which allows for dynamic allocation of memory.
- They can perform all of the same things a simple array can do and much more.

- In the `main` function, we declare `vector<double> samples(M)` which is an array of type double intialised with `M` values.
- First check you can run the results by outputting to screen

```
// we want to run M calculations
int M=100;
// now store all the results
vector<double> samples(M);
// number of paths in each calculation
int N=1000;
// run some calculations
for(int i=0;i<M;i++)
{
  cout << monteCarlo(9.576,10.,0.05,0.4,0.75,N) << endl;
}
```

download

- Now rather than output to screen write them into the vector `samples` using the array syntax like this

```
// run some calculations
  for(int i=0;i<M;i++)
  {
    // cout << monteCarlo(9.576,10.,0.05,0.4,0.75,N) << endl;
    samples[i] =  monteCarlo(9.576,10.,0.05,0.4,0.75,N);
  }
```

download

- Now rather than output to screen write them into the vector `samples` using the array syntax like this
- Calculate the mean of the vector

```
// run some calculations
  for(int i=0;i<M;i++)
  {
    // cout << monteCarlo(9.576,10.,0.05,0.4,0.75,N) << endl;
    samples[i] =  monteCarlo(9.576,10.,0.05,0.4,0.75,N);
  }
```

download

```
  double sum=0.;
  for(int i=0;i<M;i++)
  {
    sum+=samples[i];
  }
  double mean = sum/M;
  cout << " sample mean = " << mean << endl;
```

download

- Now rather than output to screen write them into the vector `samples` using the array syntax like this
- Calculate the mean of the vector
- and variance

```
// run some calculations
  for(int i=0;i<M;i++)
  {
    // cout << monteCarlo(9.576,10.,0.05,0.4,0.75,N) << endl;
    samples[i] =  monteCarlo(9.576,10.,0.05,0.4,0.75,N);
  }
```

download

```
  double sum=0.;
  for(int i=0;i<M;i++)
  {
    sum+=samples[i];
  }
  double mean = sum/M;
  cout << " sample mean = " << mean << endl;
```

download

```
  double sumvar=0.;
  for(int i=0;i<M;i++)
  {
    sumvar+=(samples[i]-mean)*(samples[i]-mean);
  }
  double variance = sumvar/(M-1);
  cout << " sample variance = " << variance << endl;
```

download

- Now rather than output to screen write them into the vector `samples` using the array syntax like this
- Calculate the mean of the vector
- and variance
- and the output is

```
sample mean = 1.28323
sample variance =
0.00609616
```

```
// run some calculations
  for(int i=0;i<M;i++)
  {
    // cout << monteCarlo(9.576,10.,0.05,0.4,0.75,N) << endl;
    samples[i] =  monteCarlo(9.576,10.,0.05,0.4,0.75,N);
  }
```

download

```
  double sum=0.;
  for(int i=0;i<M;i++)
  {
    sum+=samples[i];
  }
  double mean = sum/M;
  cout << " sample mean = " << mean << endl;
```

download

```
  double sumvar=0.;
  for(int i=0;i<M;i++)
  {
    sumvar+=(samples[i]-mean)*(samples[i]-mean);
  }
  double variance = sumvar/(M-1);
  cout << " sample variance = " << variance << endl;
```

download

- From simple statistics (Central Limit Theorem) we know that each estimate of the solution $V_N$ is

$$V_N \sim N(V^*, \sigma^2)$$

where $V^*$ is the true solution and $\sigma^2$ is the variance.

- The if we choose to take a sample mean from that distribution $\bar{V}$ with $M$ samples we have

$$\bar{V} \sim N\left(V^*, \frac{\sigma^2}{M}\right)$$

- So to get a confidence interval for our result we use this result to output the following:

```
double sd = sqrt(variance/M);
cout << " 95% confident result is in ["<<mean-2.*sd << "," << mean+2.*sd <<
    "] with " << N*M << " total paths. " << endl;
```

download

- So to get a confidence interval for our result we use this result to output the following:

```
double sd = sqrt(variance/M);
cout << " 95% confident result is in ["<<mean-2.*sd << "," << mean+2.*sd <<
    "] with " << N*M << " total paths. " << endl;
```

download

- and the output is `95% confident result is in [1.26761,1.29884] with 100000 total paths.`

# TASKS

1. Run this analysis to get confidence intervals for a range of values for $N$ and $M$
2. If $N * M$ stays the same, what can you say about the confidence interval? Why?
3. Try different payoff functions, say put options, binary options, different parameters.
4. What has the most effect on the results?
5. Have a look at the full code if you get stuck:-
   (Click here to download)

MANCHESTER 1824