Appendix E

Pragmatic C/C++

We choose to go to the moon in this decade and do the other things, not because they are easy, but because they are hard,…
John F. Kennedy

Python is slow.
https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/

When someone says: 'I want a programming language in which I need only say what I wish done', give him a lollipop.
Alan J. Perlis

Why C/C++?

In arguing for other computer languages over C/C++ a commentator wrote that the reason to learn C/C++ were limited to the following[1]

- *You absolutely need to eke out every bit of performance possible out of your software and you would like to do that with a language that will support Object-Oriented abstractions.*
- *You are writing code which will directly interface with raw hardware.*
- *Memory control and timing is of absolute importance, so you must have completely deterministic behavior in your system and the ability to manually manage memory.*

Far from persuading me to use some new *ultra high-level* "scripting-language," this pitch reminded me why I use C/C++ in the design and execution of my option investment strategies.

- For each tradable security or reference index in an options strategy there are *thousands* of options of different *strikes and maturities*. Decision making is a high-dimensional problem with computationally dense "inner-loops" that require memory and processing speed for effectiveness.
- Trading systems certainly "*must have completely deterministic behavior*," and have no use for a dynamically typed interpreted language that is notoriously intrinsically inefficient[2].
- Hedge optimization for options under real-world scenarios is a *variational problem*. The computational intensity of that, in addition to elicitation of the fat-tailed-asymmetric residual risks is in part the reason for the proliferation of risk-neutral option hedge *theories* that have disastrous practical consequences. Why should I handicap myself with an inherently slower language or one that is inefficient in utilizing computer memory?

Now that is not to say there are not approaches that could match and are perhaps more efficient than using C/C++. They involve using their precursor high-level languages (e.g., Fortran) and/or directly using machine language. The reason that C/C++ and Ada are currently used for mission critical military applications (e.g., aircraft/spacecraft and missile guidance) are *efficiency*, *performance*, and the need to have "*completely deterministic behavior*," and that they have minimal layers in between them and the hardware. These considerations are also applicable to medical imaging and geophysical imaging to aid real-time decision making.

The Avogadro Number is $6.022140857 \times 10^{23}$ and it represents the number of units in one mole of a substance. The earth's surface is estimated as 510.1 Trillion squared meters – with a Trillion being 10^{12}. The number of neurons in the human brain is estimated to be of the order 10^{11}, which is the same order of magnitude of the number of stars in the Milky Way. This gives some idea of the large-scales that need to be fathomed for computers to be applied to understand fundamentals of materials, global climate, the human brain, and cosmological mysteries. In the real-world, real-time decision-making challenges are high-dimensional. Relative to these challenging problems computers are still *puny and slow* and do not offer the luxury of inefficient[1] implementations to indulge the whim and fancy of any programing fad.

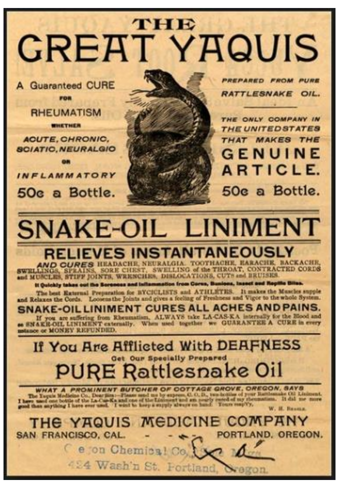


Figure E1. Next time you are pitched an *easy* information technology solution that can to solve *all problems* and has a *short development time* due to the utilization of a newfangled scripting language – think! Terseness is no substitution for completeness and false short-cuts do not empower the purported beneficiary[3].

Effective solutions require hard work to effectively articulate a viable design – irrespective of the implementation computer language. C/C++ offer implementation frameworks biased toward efficiency of processing time and memory, and minimal layers between you and your machine.

The purpose of this Appendix is to provide a display of the key C/C++ constructs used to perform the analysis in the main sections, and build portfolio monitoring and trading systems to execute the associated strategies. The sample code shown here provides a concise example based introduction to C/C++ which when supplemented by a book (or three)[4],[5],[6] can provide the reader, be they trader, quant, portfolio manager, risk-manager, an ability to implement the analysis shown in the main chapters. The purpose is also to show that C/C++ can arm you with relatively cheap computation power so that you are not held back by the notoriously ineffectual centralized information technology bureaucracies (or outright corrupt/inept outsourcing firms or the latest naïve computer science fad) to implement any ideas of yours that require significant computation. By taking matters into your own hands you gain immunity from snake charmers that offer you easy pain-free perfect solutions!

[1] Memory utilization and processing speed are the two critical limiting elements of efficiency over and above an understanding of the phenomenon being represented in a computational machine.

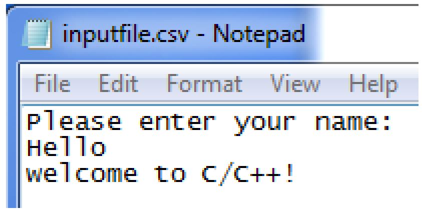
Getting Started

The sample code provided here was written on Visual Studio 2017, which implements the C++17. The coverage here provides “solved examples” that can transition a determined novice into using C/C++ for problem solving. The vastness of the combined C/C++ scope should not deter one from working with a subset that is sufficient to address one’s interests, and learning more at a slower pace or on a need to know basis. There is value in knowing the rationale for the computational problem and crafting the solution oneself, using useful subsets of C/C++.

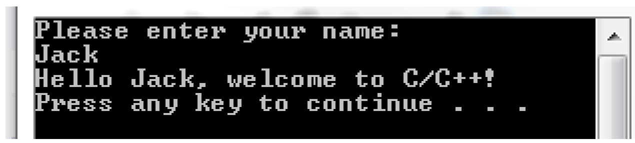
```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;
int main()
{
// Input and Output File Names
    string inputfilename    = "inputfile.csv";
    string outputfilename   = "outputfile.csv";
// Read Three Lines From Input File
    string string1, string2, string3;
    ifstream ifile(inputfilename.c_str());
    if(!ifile) cerr << "Input file is missing!" << endl;
    getline(ifile,string1);
    getline(ifile,string2);
    getline(ifile,string3);
    ifile.close();
//  Receive User Name From Console
    string name;
    cout << string1 << endl;
    cin >> name;
//  Compose Personalized Message
    string message = string2 + " " + name + ", " + string3;
//  Ouput Personalized Message on Console
    cout << message << endl;
//  Output Personalized Message in File
    ofstream ofile(outputfilename.c_str());
    ofile << message << endl;
    ofile.close();
//  Exit Console on  User Prompt
    system("pause");
    return 0;
}
```

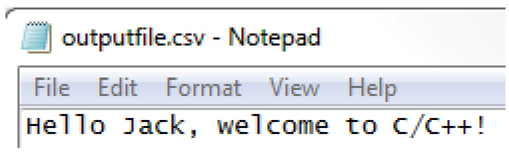
Listing1. Demonstration of File/Console Input/Output & String

File input/output is facilitated by #include<fstream>. The three rows in the inputfile.csv listed below is stored in string1, string2, and string3, line by line.



This program then produces a console that uses string1 to prompt the user to input their name which is stored in the string variable name. An output message is composed by concatenating string2, name, and string3, and is output to the console and the output file that is in the identical location as the input file.





The console waits for the user to enter any key. The input and output stream cin and cout are directly available as we preceeded int main() with #include<iostream> and using namespace std;. We are able to store words and sentences in our declared string as we have #include <string>.

I have found it useful to be able to perform file input/output using C/C++. When done with carefully designed data-structures and controlled read/write functions it can be used as the basis for a *de-facto* database (that is most likely faster than the one built by the centralized information technology bureaucracy), without the overhead of a database administrator! It has bought me White Elephant Insurance in more than one instance!

http://en.wikipedia.org/wiki/White_elephant

Figure E2. "A white elephant is a possession which its owner cannot dispose of and whose cost, particularly that of maintenance, is out of proportion to its usefulness. The term derives from the story that the kings of Siam, now Thailand, were accustomed to make a present of one of these animals to courtiers who had rendered themselves obnoxious in order to ruin the recipient by the cost of its maintenance. In modern usage, it is an object, scheme, business venture, facility, etc., considered without use or value."

Crunch Some Numbers

Computations are about inputs and processes that turn them into outputs. If the processes are organized in named functions – individually tested – it goes a long way to avoiding "spaghetti code" that is hard to re-use by the author, let alone another team member. For instance, the code in **Listing 1** would be better organized by having one function to read the inputs, another function to compose the output message, and another function to perform the output.

Having chastised myself for potentially "spaghetti code," I will clean up my act in the code listing to follow. However, let's not forget that brevity for the sake of brevity in code may not be worth pursuing religiously. Instead of pursuing brevity for showing off I recommend being content with clarity for oneself and one's colleagues. For the goal is solving the problem at hand, reliably and repeatedly, and *opening the door for solving harder problems thereafter*.

Terse code using a computationally inefficient "scripting-language" is of no use if it takes twice the memory and/or is 3 to 30 times slower for the task at hand – so much so that its proponents resort to "wrapping" computationally efficient C code to be called from it and flaunt that as its claim to fame!! What is the point of introducing lazy and inefficient code to wrap around efficient one - to win a false battle of brevity– for that precludes an evolutionary jump to solving the next level complex real problem, as the lazy language code will provide the bottle-neck.

I am not alone in recognizing this performance chasm between C/C++ and the recent breed of scripting languages. Work is underway to remedy this in newer languages that are higher level than C/C++ in their distance from the machine. Here is a performance comparison available on one such new language website:

	Fortran gcc 4.8.1	Julia 0.2	Python 2.7.3	R 3.0.2	Matlab R2012a	Octave 3.6.4	Mathe-matica 8.0	JavaScript V8 3.7.12.22	Go go1
fib	0.26	0.91	30.37	411.36	1992.00	3211.81	64.46	2.18	1.03
parse_int	5.03	1.60	13.95	59.40	1463.16	7109.85	29.54	2.43	4.79
quicksort	1.11	1.14	31.98	524.29	101.84	1132.04	35.74	3.51	1.25
mandel	0.86	0.85	14.19	106.97	64.58	316.95	6.07	3.49	2.36
pi_sum	0.80	1.00	16.33	15.42	1.29	237.41	1.32	0.84	1.41
rand_mat_stat	0.64	1.66	13.52	10.84	6.61	14.98	4.52	3.28	8.12
rand_mat_mul	0.96	1.01	3.41	3.98	1.10	3.41	1.16	14.60	8.51

Figure: benchmark times relative to C (smaller is better, C performance = 1.0).

Table E1. Performance comparison from http://julialang.org/

Gods of performance seem to have not been kind to some currently fashionable ultra-high-level languages being passed off as a panacea to the uninformed! Might they represent the building of a tower of Babel too high? After seeing such performance comparisons the question arises: Is C/C++ that hard to learn that there is a need for new languages that can *at most* match C/C++ in performance? Is not any language hard to learn initially? What if you want to solve a computationally intense problem? Which language is worth my learning time? I do not think C/C++ is as hard as the new language salesman would have you believe! Do I want a programmer building my trading system that finds C/C++ too hard? Not!

In **Listing 2** a time series is read and its mean, standard deviation, and autocorrelation is computed. The autocorrelation is computed using "brute-force," although for a latency critical real-time application I recommend using the Fast Fourier Transform (FFT). A function reads the time series – which happens to be daily data in two columns in a .csv file. The second-order statistics are calculated and output in a .csv file that the user can inspect. This mode of operation mimics a research function.

The starting point is a specific format data file that contains the data that needs to be read. Of course the data format has to be known a-priori for it to be successfully read. This data is in the file data.csv and has dates and the return separated by a ',' and a '\n' after the end of each row. The first row is a header that described the columns. The data length is not explicitly specified – a snippet is shown here. Treating this to be daily data we simply seek to extract the daily return to perform a statistical analysis on it, without any particular significance of the specific dates. The function that performs our desired task is listed right next to the data.

Date	Return
1/4/1950	0.01134002
1/5/1950	0.004736539
1/6/1950	0.002948985
1/9/1950	0.005872007
1/10/1950	-0.002931694
1/11/1950	0.003517002
1/12/1950	-0.019498402
1/13/1950	-0.005384398
1/16/1950	0.002994911
1/17/1950	0.008338345
1/18/1950	-0.000593296
1/19/1950	0.00118624
1/20/1950	0.001776725
1/23/1950	0.001182732
1/24/1950	-0.003552402
1/25/1950	-0.007142888
1/26/1950	-0.00059755
1/27/1950	0.00536514
1/30/1950	0.011820469
1/31/1950	0.001761081

```
#ifndef readtimeseries_H
#define readtimeseries_H
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
using namespace std;
void readtimeseries(string datafilename,vector<double> &data)
{
    data.resize(0);
    string dstring1, dstring2;
    ifstream ifiLe(datafilename.c_str());
    getline(ifiLe,dstring1);
    while(getline(ifiLe,dstring1,','))
    {
        getline(ifiLe,dstring2,'\n');
        data.push_back(atof(dstring2.c_str()));
    }
    ifiLe.close();
    return;
}
#endif
```

Listing 2a Sample Data and Function to Read File

It is a good idea to check whether one is indeed extracting the data as planned – by simply writng the data out in a file that can be visually inspected. By creating output at different junctures of a program one can build confidence in one's system. A portion of the check_data.csv file and the function that create it are listed here:

data
0.01134
0.004737
0.002949
0.005872
-0.00293
0.003517
-0.0195
-0.00538
0.002995
0.008338
-0.00059
0.001186
0.001777
0.001183
-0.00355
-0.00714
-0.0006
0.005365
0.011821
0.001761

```
#ifndef writetimeseries_H
#define writetimeseries_H
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
using namespace std;
void writetimeseries(string datafilename,vector<double> &data)
{
    ofstream ofiLe(datafilename.c_str());
    ofiLe << "data" << endl;
    for(int i = 0; i < data.size(); i++)
    {
        ofiLe << data[i] << '\n';
    }
    ofiLe.close();
    return;
}
#endif
```

Listing 2b Sample Data and Function to Write File

The mean and standard deviation of the data are a subset of the second order statistics. The function to assess them is listed here:

```
#ifndef calcmeanstdev_H
#define calcmeanstdev_H
#include <iostream>
#include <vector>
using namespace std;
void calcmeanstdev(vector<double> & data, double & mean, double & stdev)
{
    mean = 0.00;
    stdev = 0.00;
    for ( int i = 0; i < data.size(); i++)
    {
        mean += data[i];
        stdev += data[i]*data[i];
    }
    mean /= data.size();
    stdev /= data.size();
    stdev -= mean*mean;
    stdev = pow(stdev,0.5);
    return;
}
#endif
```

Listing 2c Function to compute mean and standard deviation of data.

The second order statistics also include how the data is correlated with itself at different lags. This ‘asynchronous correlation” of data with itself is assessed using the following function:

```
#ifndef secondorderanalysis_H
#define secondorderanalysis_H
#include <iostream>
#include <vector>
#include "calcmeanstdev.h";
using namespace std;
void secondorderanalysis(int maxLag, vector<double> & data, double &mean, double & stdev, vector<double> & autocorr)
{
    calcmeanstdev(data,mean,stdev);
    autocorr.resize(maxLag+1);
    autocorr[0] = 1.00;
    for (int iL = 1; iL <= maxLag; iL++)
    {
        autocorr[iL] = 0.00;
        for (int i = 0; i < data.size() - iL; i++)
        {
            autocorr[iL] += data[i]*data[i+iL];
        }
        autocorr[iL] /= (data.size()-iL);
        autocorr[iL] -= (mean*mean);
        autocorr[iL] /= (stdev*stdev);
    }
    return;
}
#endif
```

Listing 2d Function to compute autocorrelation of time series.

```
#ifndef write2orderstats_H
#define write2orderstats_H
#include <iostream>
#include <string>
#include <fstream>
#include <vector>
using namespace std;
void write2orderstats(string ofnss, int maxLag,  double mean, double stdev, vector<double> autocorr)
{
    ofstream ofile(ofnss.c_str());
    ofile << "lag" << ',' << "autocorr" << ',' << "mean:" << ',' << mean << ',' << "stdev:" << ',' << stdev << '\n';
    for (int i = 0; i <= maxLag; i++)
    {
        ofile << i << ',' << autocorr[i] << '\n';
    }
    ofile.close();
    return;
}
#endif
```

lag	autocorr	mean:	0.000293	stdev:	0.00971
0	1				
1	0.028597				
2	-0.04012				
3	0.002085				
4	-0.0071				
5	-0.0119				
6	-0.00561				
7	-0.01874				
8	0.009866				
9	-0.00611				
10	0.011997				

Listing 2d Function to write second order statistics of time series and sample output.

The different modules that are used to solve the problem of computing the second order statistics are shown above. These functions are easy to read and can be tested on a stand-alone basis. They are assembled in the main program that sequentially orchestrates these functions:

```
#include <iostream>
#include <vector>
#include <string>
#include "readtimeseries.h"
#include "writetimeseries.h"
#include "secondorderanalysis.h"
#include "write2orderstats.h"

using namespace std;
int main()
{
//  I/O Filenames
    string ifn = "data.csv";
    string ofn1 = "check_data.csv";
    string ofn2 = "secondorderstats.csv";
//  Read/Write Time Series
    vector<double> data;
    readtimeseries(ifn,data);
    writetimeseries(ofn1,data);
//  Compute and Write 2nd Order Statistics
    int maxLag = data.size()/10;
    double mean, stdev;
    vector<double>autocorr(maxLag+1);
    secondorderanalysis(maxLag,data,mean,stdev,autocorr);
    write2orderstats(ofn2,maxLag,mean,stdev,autocorr);
//  Exit Console on User Prompt
    system("pause");
    return 0;
}
```

Listing 2e Main program to compute second order statistics of time-series. This listing demonstrates data types int and double, the vector container, and use of functions and passing data by reference and values to functions. The use of the for loop and nested loops are also shown, as is the parsing of .csv files.

The main program is quite succinct, as are each of the functions included in the main. Only one 'computation' is done in the main – assessing the maximum lag of autocorrelation to be the integer near 100th of the number of data point. Perhaps a function could be usefully written for that too and maxLag could be an exogenous user input (via console or file). To prevent the console application from quickly scrolling and terminating the user is prompted for a keyboard input.

In a real application the data could be real-time and/or sourced from an EMS or Bloomberg and the computed second order statistics could be converted into a trading signal and linked to a portfolio management tool that perhaps has a trade generator. In those applications some other function may take in the second order statistics as parameters, instead of or in addition to a file output, which might still be used to provide a record of the second order statistics being used to make a decision.

Monte-Carlo Simulation

We saw some random looking data in the last example and found its mean, standard deviation and correlation with itself over different time-lags. The correlation seemed to die sharply. These quick observations – juxtaposed with *beliefs* about *efficient markets* - serve as the basis for random-walk model of an asset – where the returns are assumed to be independent over distinct time-steps. In the main section we show that this is an unsatisfactory framework for describing asset returns and understanding risk-return opportunities in options. Here we simply demonstrate an implementation of the random-walk model. The implementation framework can then be extended to more realistic models of an asset – like the one made in the main section.

In the random walk model the mean and standard deviation of the return characterize the Normal distribution of the returns that are assumed to be independent over the different time steps. A sequence of identically and independently distributed random Normal random variates yields a time-series. The notion of an *ensemble* is central to Monte-Carlo simulation. The return time series describes one possible outcome (sometimes referred to as 'path') among the ensemble. In certain applications it is useful to assess the uncertainty over the ensemble – driven from the uncertainty in random returns. So we will demonstrate generating an ensemble of return time-series. We will employ a canned random number generator [7] and specify the length of the time series as well as the number of realizations needed. We will implement assessments of simulated statistics via the statistics of one time series or that of the ensemble at different time steps. This problem is mathematically simple – however it has sufficient complexity to illustrate the power of C/C++ and the utility of user defined objects via classes.

Let us call this class whitenoise. We can instantiate a whitenoise object with a name of our choice – say mywhitenoise. We need to specify how many time steps we had in mind, and how many random paths we need to perform some statistical analysis. The simulated process is characterized by a Normal Density that in turn is characterized by its mean and standard deviation. This provides an example of how a concept creates an ecosystem that is usefully recognized as a user-defined type – an object of a user-defined class. This facilitates a higher-level language that can be used to marshal complex objects. This is useful to some extent – as long as one does not forget the building blocks – and waste memory and processing time.

```
#ifndef GUARD_whitenoise_H
#define GUARD_whitenoise_H

#include <iostream>
#include <vector>

#include "..\\..\\..\\..\\utilityE\\nrcpp\\nr.h"
using namespace std;

class whitenoise
{
private:
//  members
    double mean;
    double stdev;
    int numsteps;
    int numrLzns;
    vector<double> simstat;
    vector<vector<double>> noise;
//  methods
    void calcstats();
public:
//  constructors
    whitenoise();
    whitenoise(double _mean,double _stdev,int _numsteps, int _numrLzns);
//  destructor
    virtual ~whitenoise();
//  accessor
    double get_mean();
    double get_stdev();
    int get_numsteps();
    int get_numrLzns();
    double get_sim_mean();
    double get_sim_stdev();
    double get_sim_skewness();
    double get_sim_kurtosis();
    vector<double> get_path(int _irLzn);
    vector<double> get_ensemble(int _it);
    vector<vector<double>> get_ensemble();
//  mutator
    void set_mean(double _mean);
    void set_stdev(double _stdev);
    void set_numsteps(int _numsteps);
    void set_numrLzns(int _numrLzns);
//  methods
    void showparameters();
    void simulate(int _rseed);
    void showsimstats();
    void write_path(string _filename, int _irLzn);
    void write_ensemble(string _filename, int _it);
};
#endif
```

Listing3a. Class Declaration for “whitenoise.”

```
#include "whitenoise.h"
void whitenoise::calcstats()
{
    simstat.resize(4,0.00);
    int datasize = (this->numsteps)*(this->numrLzns);
    double datapoint,pertdatapoint;
    for (int it = 0; it < this->numsteps; it++)
    {
        for (int irLzn = 0; irLzn < this->numrLzns; irLzn++)
        {
            datapoint = this->noise[it][irLzn];
            simstat[0] += datapoint;
            simstat[1] += datapoint*datapoint;
        }
    }
    simstat[0] /= datasize;
    simstat[1] /= datasize;
    simstat[1] -= (simstat[0]*simstat[0]);
    simstat[1] = pow(simstat[1],0.5);
    for (int it = 0; it < this->numsteps; it++)
    {
        for (int irLzn = 0; irLzn < this->numrLzns; irLzn++)
        {
            pertdatapoint = this->noise[it][irLzn] - simstat[0];
            simstat[2] += pertdatapoint*pertdatapoint*pertdatapoint;
            simstat[3] += pertdatapoint*pertdatapoint*pertdatapoint*pertdatapoint;
        }
    }
    simstat[2] /= datasize;
    simstat[3] /= datasize;
    simstat[2] /= (simstat[1]*simstat[1]*simstat[1]);
    simstat[3] /= (simstat[1]*simstat[1]*simstat[1]*simstat[1]);
}
whitenoise::whitenoise()
{
}
whitenoise::whitenoise(double _mean,double _stdev,int _numsteps, int _numrLzns)
{
    mean        =   _mean;
    stdev       =   _stdev;
    numsteps    =   _numsteps;
    numrLzns    =   _numrLzns;
}
whitenoise::~whitenoise()
{
    noise.clear();
    simstat.clear();
}
```

Listing3b. Class Implementation for "whitenoise."

```
double whitenoise::get_mean()
{
    return mean;
}
double whitenoise::get_stdev()
{
    return stdev;
}
int whitenoise::get_numsteps()
{
    return numsteps;
}
int whitenoise::get_numrLzns()
{
    return numrLzns;
}
double whitenoise::get_sim_mean()
{
    return this->simstat[0];
}
double whitenoise::get_sim_stdev()
{
    return this->simstat[1];
}
double whitenoise::get_sim_skewness()
{
    return this->simstat[2];
}
double whitenoise::get_sim_kurtosis()
{
    return this->simstat[3];
}
vector<double> whitenoise::get_path(int _irLzn)
{
    vector<double> path(this->numsteps);
    for (int it = 0; it < this->numsteps; it++)
    {
        path[it] = noise[it][_irLzn];
    }
    return path;
}
vector<double> whitenoise::get_ensemble(int _it)
{
    return noise[_it];
}
vector<vector<double>> whitenoise::get_ensemble()
{
    return noise;
}
void whitenoise::set_mean(double _mean)
{
    mean   = _mean;
}
void whitenoise::set_stdev(double _stdev)
{
    stdev = _stdev;
}
```

Listing3c. Class Implementation for “whitenoise.”

```
void whitenoise::set_numsteps(int _numsteps)
{
    numsteps = _numsteps;
}
void whitenoise::set_numrLzns(int _numrLzns)
{
    numrLzns = _numrLzns;
}
void whitenoise::showparameters()
{
    cout    << " White-Noise Parameters:" << endl;
    cout    << " mean:      "  <<    this->mean      << endl;
    cout    << " stdev:     "  <<    this->stdev     << endl;
    cout    << " numsteps: "   <<    this->numsteps  << endl;
    cout    << " numrLzns: "   <<    this->numrLzns  << endl;
}
void whitenoise::simulate(int _rseed)
{
    noise.resize(0);
    vector<double> ensemble(this->numrLzns);
    for (int it = 0; it < this->numsteps; it++)
    {
        for (int irLzn = 0; irLzn < this->numrLzns; irLzn++)
        {
            ensemble[irLzn] = this->mean + this->stdev*(NR::gasdev(_rseed));
        }
        noise.push_back(ensemble);
    }
    calcstats();
}
void whitenoise::showsimstats()
{
    cout << " Statistics of Simulated Noise" << endl;
    cout << " mean     : " << simstat[0] << endl;
    cout << " stdev    : " << simstat[1] << endl;
    cout << " skewness : " << simstat[2] << endl;
    cout << " kurtosis : " << simstat[3] << endl;
}
void whitenoise::write_path(string _filename, int _irLzn)
{
    ofstream ofile(_filename.c_str());
    ofile << "step" << ',' << "noise" << '\n';
    for (int it = 0; it < this->numsteps; it++)
    {
        ofile << it << ',' << noise[it][_irLzn] << '\n';
    }
    ofile.close();
}
void whitenoise::write_ensemble(string _filename, int _it)
{
    ofstream ofile(_filename.c_str());
    ofile << "path" << ',' << "noise" << '\n';
    for (int irLzn = 0; irLzn < this->numrLzns; irLzn++)
    {
        ofile << irLzn << ','  << noise[_it][irLzn] << '\n';
    }
    ofile.close();
}
```

Listing3d. Class Implementation for "whitenoise."

```cpp
#include <iostream>
#include "whitenoise.h"
using namespace std;
int main()
{
    double  mean   = 0.000293;
    double  stdev = 0.00971;
    int     numsteps = 252;
    int     numrLzns = 10000;
    int     rseed = -1234;
    int     it = 0;
    int     irLzn = 0;
    string  pfilename = "path.csv";
    string  efilename = "ensemble.csv";
    whitenoise mywhitenoise(mean,stdev,numsteps,numrLzns);
    mywhitenoise.simulate(rseed);
    mywhitenoise.showparameters();
    mywhitenoise.showsimstats();
    mywhitenoise.write_path(pfilename,irLzn);
    mywhitenoise.write_ensemble(efilename, it);
    system("pause");
    return 0;
}
```

Listing3e. Main demonstrating "whitenoise."

```
K:\vsoftw\test\C-C++basics\MC\Release\MC.exe
 White-Noise Parameters:
 mean:      0.000293
 stdev:     0.00971
 numsteps: 252
 numrLzns: 10000
 Statistics of Simulated Noise
 mean     : 0.000287536
 stdev    : 0.00970929
 skewness : -0.00157719
 kurtosis : 3.00257
Press any key to exit:
```

How to Chew More Gum While Chewing Gum

Often accomplishing a task requires performing independent computations and assembling their results. These independent computations could involve overlapping inputs, or can be non-overlapping inputs.

An example of an *embarrassingly parallel* computation problem is simulating a large number of random numbers. Since we will be making console output to monitor performance and storing the random numbers in a vector, we make the following inclusions:

```
#include <iostream>
#include <vector>
#include <ctime>
#include <thread>

using namespace std;
```

The rather simple operation, that we would like to perform lots of, is encapsulated in a function.

```
void genURV(int iseed, int n, vector <double>& rn)
{
    srand(iseed);
    rn.resize(n);
    for (int i = 0; i < n; i++)
    {
        rn[i] = (double)rand() / RAND_MAX;
    }
    return;
}
```

Admittedly the case for multi-threading becomes stronger if the function were more time-consuming and complex.

The problem is introduced using a single thread:

```
int N = 800000000;
clock_t startTime, endTime;
double timetaken;
cout << "Single Thread Example" << endl;
startTime = clock();
vector<double> rn;
int iseed = startTime;
genURV(iseed, N, rn);
endTime = clock();
```

This is followed by the multithreaded example that employs 8 threads and tasks each one of them with 1/8th the work.

```
cout << "Multi-Threaded Example" << endl;
startTime = clock();
int NTH = 8;
int NPTH = N / NTH;
vector<thread> mythreads;
vector<double> thrn(N);
vector<vector<double>> ithrn(NTH);
int iseed2;
for (int i = 0; i < NTH; i++)
{
    iseed2 = (int)clock();
    mythreads.push_back(thread(genURV, iseed2, NPTH, ref(ithrn[i])));
}
cout << "threads spawned" << endl;
for (int i = 0; i < NTH; i++)
{
    mythreads[i].join();
    for (int j = i * NTH; j < i*NTH + NTH; j++)
    {
        thrn[j] = ithrn[i][j - i * NTH];
    }
}
endTime = clock();
```

The listing shows how to create threads and wait for them to be done before proceeding ahead. The computation time is measured to provide a tool to examine if multi-threading is indeed helping you. There is overhead of creating threads and also of assembling the results of each thread into an overall data structure. That is why the processing time is not inversely related to number of threads.

```
K:\vsoftw\applications\Book\x64\Release\ChewGumWhileChewingGum.exe
Single Thread Example
time-taken: 20.443 seconds
Multi-Threaded Example
threads spawned
time-taken: 6.976 seconds
Press any key to continue . . .
```

Speed Writing and Reading

In certain applications we might be required to write large quantities of data into files and read the files. Here we present a specific example of reading and writing uniformly distributed random numbers. To store these numbers in a vector and create files and subsequently read from them we include some basic utilities:

```
#include <iostream>
#include <vector>
#include <fstream>
#include <ctime>
#include <string>

using namespace std;
```

A vector of random numbers is created for our demonstration of binary files and comparison with .csv files.

```
int N = 10000000;
vector<double> rn(N);
for (int i = 0; i < N; i++)
{
    rn[i] = (double)rand() / RAND_MAX;
}
```

Writing this vector in a ‘csv file (single column so no comma needed!) was described here earlier. It simply is as follows:

```
string ofn = "rand.csv";
ofstream ofile(ofn.c_str());
for (int i = 0; i < rn.size(); i++)
{
    ofile << rn[i] << '\n';
}
ofile.close();
```

Instead of directly writing the number we can point a char pointer to the double and write them in a binary file:

```
string ofn = "rand.bin";
ofstream ofile(ofn.c_str(), ios::binary);
for (int i = 0; i < rn.size(); i++)
{
    ofile.write((char*)&rn[i], sizeof(double));
}
ofile.close();
```

The reading of a csv file described before involved converting the read string into a floating point

```
vector<double> rrn;
ifstream ifile(ofn.c_str());
string dstring;
while (getline(ifile, dstring, '\n'))
{
    rrn.push_back(atof(dstring.c_str()));
}
ifile.close();
```

The reading of the binary file involves slicing the *buffer* by an amount required to store a double

```
vector<double> rrn;
ifstream ifile(ofn.c_str(), ios::binary);
char buf[sizeof(double)];
while (ifile.read(buf, sizeof(double)))
{
    double* temp = (double*)buf;
    rrn.push_back(*temp);
}
ifile.close();
```

The time taken for these distinct methods can be compared using the clock. At the start and end of the tasks we log the time and output the difference:

```
clock_t startTime = clock();

clock_t endTime = clock();

double timetaken = (endTime - startTime) / (double)CLOCKS_PER_SEC;
```

We have all the parts to demonstrate the efficiency of binary files versus .csv files. In the example problem above I/O using a csv file is 200% more time consuming than I/O using a binary file.

```
K:\vsoftw\applications\Book\x64\Release\binaryIO.exe
Time taken to write and read 10000000 random variates in binary: 8.924 (secs).
Time taken to write and read 10000000 random variates in csv: 27.93 (secs).
Press any key to continue . . .
```

Afterword

The samples here covered a selective journey through C/C++. Nevertheless, if you can follow and replicate the examples provided here, then you are capable of harnessing the power of C/C++ to implement the analysis shown in other sections. You are also prepared to see other vistas and use C/C++ to solve problems of your choice.

References

[1] http://simpleprogrammer.com/2012/12/01/why-c-is-not-back/

[2] https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/

[3] *Straight Talk In the Debate Over Ebonics by* Clyde Haberman New York Times Dec 31 1996. https://www.nytimes.com/1996/12/31/nyregion/straight-talk-in-the-debate-over-ebonics.html

[4] *The C Programming Language, 2nd edition*, Brian W. Kernighian and Dennis M. Ritchie, Prentice Hall 1988.

[5] *The C++ Programming Language, 4th edition*, Bjarne Stroustrup, Addison Wesley, 2013.

[6] *Professional C++4th edition*, Marc Gregoire, John Wiley & Sons 2018.

[7] *Numerical Recipes in C++: The Art of Scientific Computing, 2nd edition*, William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, Cambridge University Press, 2002.