Learning C++: A Short Novel-In-Progress

--Industry Trends, Standards & Best Practices
Version 0.02.1

-By R.A. Nagy
President, Soft9000.com

Forward

Welcome to the wonderful world of C++ !

In a time when computer languages self-deprecate more and more each day; During an era when computers no longer double processing power every other year... today more than ever people are looking for faster, more efficient, secure and longer-lasting programming languages.

Indeed, while I personally love writing software in C#, VB.NET, and Java, because these languages are easy to reverses-engineer, I ***never*** write anything I *plan to sell* in them. Why? Because when we write in .NET and Java, even *novice* software developers will find it far too easy to plagiarize[1] our code... let alone our data!

So add to what might yet become a *screaming need* for faster and more enduring software, a *financial craving* to create software that *protects your intellectual property*, and you now understand the need to understand modern C/C++. There is, after all, a *reason why* all popular operating systems – as well as many programming languages themselves - are written in C/C++!

Approach
By reviewing how to create useful, real-world programs, our goal is to gift you tools that you can use right-away. -Within the first few chapters the reader will learn how to create file-based data-collection programs, write logging / note taking software, and even display report that you can view in a browser.

After completing the training on Soft9000.com you will understand enough C++ to do even more: You will be able to understand how to use tools that can send and receive electronic mail, read and post Usenet Articles, manage attachments, and even interface with external programs such as Sendmail & FTP.

Finally, in addition to real-world examples, the beginning of this book offers the reader an easy-to-read, enjoyable tale. One designed to allow you to create and understand an original set software carefully crafted to introduce you to powerful conventions like templates, namespaces, Internet RFCs, STL, HTML, design patterns, and a whole lot more. Along the way, we will discuss crucial programming concepts. Topics such as naming conventions, testing techniques, and the importance of standards. Along the way we will even present a unique, personal software development methodology. Motivational and programming ideas that you will need to ensure that your software stays

1 The problem with .NET and Java has nothing to do with the languages written *per se*. The problem lies within their use of unsecured 'byte code'. Because byte-codes have been designed to be as portable as possible, byte code is far easier to reverse-engineer. Once you have a program, byte code is a lot easier to generate source from that via an equivalent amount of platform-specific *machine code*. -The very type of code always generated by C/C++.

as good as when you first wrote it.

All told, this tome offers the reader a truly unique learning experience. A journey designed to quickly familiarize you with key language concepts, software, and professional developer techniques. Tools & concepts that we will need to tackle real-world programming chores, ber they at-home or on-the job. -Far from showing you merely how to program C++, "The Joy of C++" is here to teach you how to become a successful software developer. Today and always!

--Enjoy the Ride!

R.A. Nagy

Table of Contents

Day One: Working Late

While working late one night at a New York Metropolitan Library, I was suddenly forced to wake up from my deep pondering of an outline. Suddenly, seemingly exploding out of nowhere, a lighthearted male voice proclaimed “What an odd name for a book!”

After discovering that my heart had indeed not stopped, I wryly looked up from the late night diversion into a familiar face. It was the face of Dave, a friend from work.

While somewhat lacking in tact, when it came to long-term industry tenure Dave was one of those guys you just love to talk to. From Punch Cards to PDA's, Dave has seen a lot of software and hardware come and go. Instantly united by our ridiculously long terms in the computing industry, we often swapped insightful Dilbert, Gates, and industry concepts at lunchtime.

“You think so?” I replied, donning a smile.

“Yes” added a female voice from just behind my right shoulder. “It will never sell!”

“There are more reason to write a book than for fame and money” I retorted automatically.

“Good.” He said. “You might just have another one there!”

We laughed.

Turning to confirm the face that threatened a second thread, I scanned the anticipated mass of bollowing blond hair. -After confirming the source of the utterance, Dave's retort also seemed to have set her in motion from behind me.

“Hello Debbie” I said, following her prance around the right side of the table. After she came to rest upon the left-hand side of Dave, grinning broadly I concluded “Odd to see you both here after work!”

“We’ve been stalking you” Dave smiled, glancing a little craftily at Debbie. “Yes” she said, placing her hands on her hips, “We wanted to see what a principal trainer did after work!”

“Well”, I mused out loud “If they want to keep themselves marketable, then they do not spend too much time watching television!”

“I like watching TV” said Debbie.

“Yea” said Dave. “Where else can anyone learn so much about stalking?”

Chuckling, I retorted, “I stalk thee, wild project!

Project Stalkers

“Like many, in my spare moments I usually like to hunt projects. Interesting software is my game. -As for tonight however, it just so happens that - a few nights back - I decided to share what I have learned about industry trends, standards, best practices, and C++.”

“That’s sick” Dave chuckled, smiling in such a way underscore his intonation that he meant the exact opposite of what he said. He then looked at Debbie and said, in his best behaved voice “Maybe that is what I have been doing wrong all these years!”

“I'd rather see cool projects on a resume, than a moose-head on a wall” Debbie laughed.

“When it comes to programming for its own sake, no one can do anything wrong” I replied. “Practice alone makes perfect. -What we do for a living determines what living we can do.”

“That’s deep” -said Debbie. “How many years did it take to arrive as that level of ‘nerdvanna?’”

“Never left it!” I said. “Geekiness is a state of being.”

We laughed again.

Noticing that randomly bebooked heads had been increasingly turning our way, we decided to relocate into one of the more riot-resistant conference rooms.

After settling in, looking at Debbie, Dave theorized “Practice makes perfect; we either like what we do, or we do not. After committing to a task, onlookers who are not enjoying things as much as we are then put the ‘geek’ or ‘nerd’ label on us. My theory is that the do that so they know to keep out of arms reach.”

Laughing again, each of us nodded appreciatively. United as we were at that night within the towering library book stacks - happily sequestered as we were behind the cold stone lions of Manhattan - there could be no denying that we were technology geeks… and we knew it.

The Source for All Seasons

Yes, we were indeed geeks: While Dave could tell you how many dust bunnies there are in the average server room, as our resident Java ‘Gurett, Debbie could accurately calculate the rate deprecation of her software to nine decimal places. But unlike Dave and I, Debbie had been writing software for less then 3 years. Evolving from a mainframe background, Dave had just begun to learn Object Orientation using C#. Dave also knew

C.

“I have always wanted to learn C++,” Debbie said brightly. Then looking over at Dave to see the he was also nodding his head in agreement Debbie added, “maybe we will buy your book!”

“Well, if my mother purchases a copy, then that will be at lease three sales” I chuckled. “Why not help me write it?”

“You wouldn’t want us to do that” said Dave. “We inadvertently signed away our rights to everything that we *ever* write. If we help you, our company would own your book!”

“Okay, four sales, then” I chuckled.

Then adding a little more seriously I said “Then why not learn C++ while I write the book” I thought out loud. “We can meet here in the evenings two or three times as week. After our lessons I will transcribe what happens into the book. That will make for a very good rough draft. -After I spice things up a bit, not only will you both know C++, but you can help me turn this technical outline into an instructional novel!”

No sooner than I had spoken these words, we all felt a strange glow about the place. Not only did the library suddenly seem just a little brighter, but we all also felt slightly more energetic.

“Why not?” Said Debbie

“How bout we start tonight?” asked Dave.

“Okay” I said. "I am as ready to start teaching you what I have learned tonight."

Pointing to a quiet room on the eastern side of the library, Dave said, “There is a screen projector in that room. Let’s work in there.”

Basic Program Structure

After settling ourselves yet again into a slightly sweat-smelling workroom, I rummaged around my hard drive looking for some of my articles. After ‘netting them to their email addresses, I opened the source code from one and displayed it on my laptop:

```
#include <iostream>

int main(int argc, char *argv)
{
   std::cout << "Hello world!";
   return 1;
}
```

“This is a typical C++ program” I said. “Notice that the program begins with a function called ‘**main**’. The function begins at the first curtly brace, and ends with the last.

Since lower and upper case characters are as different in C++ as they are in other modern languages, the fact that the main starting point is lower case allows us to do things like create a “**Main**” function somewhere else as well. If we wanted to we could make our program look and operate much like they do in Java and C#. -But both C and C++ will only start at “**main**”.

Introduction to Namespaces

Dave asked "What is that rather intimidating '**std::**' prefix all about?"

```
   std::cout << "Hello world!";
```

"The name on the left – the **std** before the '**::**' - refers to a common *collection* of previously written software. Known as a **namespace**, in this case **std::** is a prefix that we us to access the **std Namespace**.”

Looking ad Debbie, I explained “Much like packages Java, in C++ a Namespace represents large set of things we can re-use. A set of vendor-independent, standards-based functions, the **std** Namespace includes a way to work with files, foreign languages, collections, and more.”

Using “using”

"In addition to leveraging namespaces created by others” I continued “over time the goal of many tenured C/C++ software developers is to create their own namespaces, as well. If for no other reason than for personal use, having a place to store the things you like to use most makes future software creation a lot faster.

Because typing the name of a namespace (in this case, “**std::**”) all the time can become tiresome, if we add a **using statement**, then we do not have to keep on adding the namespace-name in front of everything.”

I quickly updated the program to demonstrate how namespaces eliminate the need to add the **std** namespace qualifier before **cout**:

```
#include <iostream>

using namespace std;

int main(int argc, char *argv)
{
   cout << "Hello world!";
```

```
    return 1;
}
```

“Once we have included a namespace into our project, we can always re-use a well-known collection of capabilities. In the above example, we are re-using **cout** from the **std**.**”** namespace.

Both Dave and Debbie were familiar with the concept. Because both C# and Java have an equivalent, they both quickly understood how having keywords such as “**using**” can save us a lot of typing.

"By adding a using statement, we can eliminate the need to type the namespace prefix.” I added “The only time we would have to provide that prefix again would be when the compiler cannot find a declaration, or in a rare instance when we are **using** two namespaces that each contain something having an exact same member-name.”

“By member-name you need to tell everyone that you are referring to something underneath a top-level name” Dave said. “That confuses a lot of people.”

“Good point” I said. “If we were using two namespaces that each had a **cout** for example, the compiler would need some way of resolving what many compilers call an *ambiguity*. In this case, an ambiguity occurs when two or more namespaces have something that is named the same."

The Pre-Processor

"Easy enough to understand." said Debbie. "What does that # sign do?”

“The 'pound sign' is a special symbol. It is detected by one of the most unique tools of the C/C++ Language. -Used to tell us that we are talking to another tool, lines prefixed by # are interpreted *just before* our program is translated into machine-readable form.”

“-Compiling” Dave added, translating my flare for too many words into something that Debbie could better relate to.

“Got that, thanks” smiled Debbie. “I would love to have the ability to do that in Java. Much of what slows us down is having our code translated from bytecode into machine readable form.”

“It can be much the same for .NET” said Dave. “While we can create machine-code, most folks do not feel the need to do so. Even when we use .NET natively, we have lots of slow dependencies. Components that the computer has to spend a lot of time locating, and loading.”

“GAC” I exclaimed, in a cough-like voice.

Dave translated our laugh for Debbie[2].

Continuing, I explained “Because anything that has a # in front of it can be translated *before* it is compiled, C++ gives software developers a chance to manage things before a compiler works its magic. In this case, that “**#include**” -step tells a tool to locate, and insert, another file.

Known as the Pre-Processor, as its name implies, here it is being told to include, or insert, a file at the present location.

While **#include** works something like the ***import*** statement in Java, in this case we are using the pre-processor to *insert* a file. Unlike other programming languages, the pre-processor can also define a macro, global variable, or include another file, anyplace in our source code.”

“That could get confusing” Dave said.

“Perhaps” I admitted, “but it can also come in handy. The flexible to use other “#” statements – such as **#if**, **#else**, and **#define** allows us to arrange code to do things *before* a compiler touched our software. While over-using the pre-processor can indeed become a problem, proper usage can allow use to do lots of very creative things. Indeed, much like using pointers, the Pre-Processor is why programs written in C++ are usually faster, and smaller, than those written in every other programming language.”

More Pre Processor Examples

“How can using a pre-processor make things faster?" Debbie asked.

Quickly typing, I added a few more lines of programming:

2 The “Global Assembly Cache” is often one of the most abused features of .NET. Interested parties can 'google the term “DLL Hell” (anonymously coined in the December 1993 edition of BYTE Magazine) to see what problems the GAC was designed to remedy.

```
#include <iostream>

using namespace std;

#define SAY_HELLO

int main(int argc, char *argv)
{
#ifdef SAY_HELLO

   cout << "Hello world!";

#else

   cout << "Welcome to C++!";

#endif

   cout << " Please type a number and press enter: ";

   int iNum = 0;
   cin >> iNum;
   cout << endl;
   cout << "You typed: " << iNum;

   return iNum;
}
```

Saving the program as **hello2.cpp**, I said "Look at those **#define**, **#ifdef**, **#else**, and **#endif** statements. The indentation clearly shows that they are delimiting blocks – or an area containing many individual statements - of code.

"Because the pre-processor will allow us to assemble key blocks of code *before* a program is compiled, we do not have to take up valuable computing real-estate. Rather than loading code that we will never use, we can simply exclude it.

When we can completely remove any unnecessary code without referencing a labyrinth of external library invocations - the pre-processor is the first reason why C and C++ programs can be so much more efficient."

C / C++ Efficiencies

Looking at Debbie, Dave observed "The smaller the program, the faster it downloads and runs. No wonder every major operating system is written in C and C++!"

"Well, Assembly Language can creep in from time to time as well. Much the same way that C/C++ creeps into .NET and Java" I added, "But the pre-processor works the same for both the C, and C++, Languages.

"Pre-processors let us do a lot more before our software is compiled. In the hands of an

experienced software developer, there is no need to fear them... or things like pointers. But Dave is right – in the hands of an inexperienced software developer, the misuse of such raw power can quickly get others into trouble. Software can become virtually impossible to maintain. One of the reason behind Java was to protect others from the abuse of such powerful features."

"Yea" said David, then looking a Debbie, he said pointedly "Like having our programs crash because of null pointers..." Frowning comically, without looking at him Debbie sharply replied "Well, Microsoft copied that one[3], too!"

Clearing my throat somewhat loudly, I pressed on "Before the advent of Object Orientation, using #define and #ifdef was how many C programmers created software that could be used across several operating environments. Indeed, one of the reasons why I enjoy C++ is because I can write programs faster across many operating systems. When it comes to open source, using the pre-processor to include WINDOWS or LINUX code is a lot easier than creating & shipping a lot of libraries."

"In many Open-Source projects" Dave added "we should be on the lookout for pre-processor definitions that look like WIN32', UNIX, OSX, DOS, and more. Seeing pre-processor definitions such as those tell us what to expect."

"Does anyone sell a compiler for all of those operating systems?" Debbie asked.

"Undertakings such as GNU, Microsoft, Intel, IBM, and others have traditionally had their own C++ Compilers" I said. "In response to the free software movement, many companies even offer free versions of their compilers, as well as other programming tools[4].

But when it come to sheer ubiquity, by far the most comprehensive support I have seen has been for the GNU C Compiler (GCC). -While certainly not the fastest, compliant, or efficient compiler across *all* operating systems, GCC is certainly the most readily available."

Writing Software Once ... and for All!

"Having lots of independent compiler creators is another reason why geeks like Mr. Nagy here like C++" Dave added. "Oracle owns Java. Microsoft owns C# and .NET. Because these companies are the owners of those languages, they are able to decide the future of our software."

---

3 One of the early justifications behind the creation of Java was to make programming easier for the masses. Key amongst the interesting claims was the intent to eliminate all dangers of using *null* – also arguably known as *uninitialized* or simply *invalid* – pointers. It did not happen.

4 Robotics enthusiasts have noted that from the Basic Stamp to the Z-8000, that C/C++ tools also abound. I myself am a fan of the Atmel AVR series of microcontrollers. The GCC works well there, as well.

"Yes!" exclaimed Debbie. "I am so frustrated when a feature I have relied upon is deprecated. It always makes you feel so dirty. Deep down I know that is just a matter of time before I will have to re-write everything."

RANT: On Standards

"Exactly" I said. "Because C++ is OPEN-standards based, unlike other object oriented languages, change take place only after years of consideration. Much like the Internet Standards themselves[5], when you maintain an open standard everything is subject to a LOT more scrutiny.

"In the case of C/C++, people who have invested millions of dollars in computing have a chance to participate in the future of the C/C++ Language Standard. Even before the rise of Object Orientation, the universal availably of standards such as the **Standard C Library** is what helped propel many innovative technologies – Like Internet Browsers, CGI, and PHP – into super-stardom. I rely upon the Standard Library in my Open Source C/C++ Projects.

"When we use world-wide standards, unlike the internal machinations that take place behind closed doors at many companies, people like you and I can choose to become active in the standards creation process. Of course, the fact that everyone can comment is one of the reasons why it takes more time to maintain any *truly* open standard. -But a standardization process allows those who have the most investment dictate our needs to vendors, rather than the other way around.

"The standardization process is why programming languages like C++ live for decades. -People do not sit around plotting how to get us to buy more products; Companies do not try to lock us into their technologies. Finally, because open standards are motivated more by stability, and less by profit, standardization committees take the time required to ensure that change disrupts investment as little as possible."

Snorting, Dave added: "Yea – From MFC or ATL, to WinForms, to WPF, when is it going to sink-in that SOME companies enjoy throwing away technologies, far more than they enjoy creating them?"

"That may be true" I replied. "Yet even Open Source language like PHP leave a lot of folks behind. The shift from PHP 4 to PHP 5 was a real tragedy for many.

Yet be the switch to standards be IBM coining AIX, or Apple adopting OS X... look how profitable REAL technical-savvy folks have found C/C++ to be! By not requiring us to

---

5 The standards that manage the Internet are most often distributed as a Request For Comment, or **RFC**. Linux has a set of open standards known as **POSIX**. Indeed, even open, yet proprietary languages – such as Java – have a community process. Known as the "Java Specification Request", or **JSR**. Rather than conveniently obsoleting technologies in the hopes of licensing completely new ones, open communities allow everyone to participate in the future of the technologies they have invested in.

do things over again – let alone to spend industry *billions* on training, tooling, and new licenses - it is indeed possible to have our hardware and software investments keep-on saving us money...!

So while easy, fast, cheap, & disposable technologies certainly have their place, when we have to use them, determining *how long we can use them* is not something that I want OTHERS to have control of."

RANT: The "Free Software Threat"

"No wonder companies do not like standards." said Dave "Not only does the standardization process take time, but open standards take them out of the drivers seat. Rather then allowing any single company to dictate the future of our software, open standards make companies *compete with each other* to keep *us* happy."

"Having served as a Principal at a few companies," I said: "I can tell you that there is a lot of fear at the prospect of competing with freely available tools. But such fears usually overlook the need for staying power!"

"What does THAT mean?" asked Debbie.

Continuing, I observed "Giving things away does not pay the bills. Unless you can find financial support from a company that actually *sells* something, then the staying power of 'free' is pretty much non-existent. If we are pondering the free software model, we will need to define 'success' in other terms. We must also be prepared to let well-known companies make lots of cash from our efforts."

"How can companies make money from your project?" asked Debbie. "Doesn't Open Source licenses prevent that?"

"A licenses is only as good as the will to enforce it" Davie interjected.

Reverse Engineering

"Good point" said I. "Other companies can also profit by your efforts by clean-rooming[6] your code. Indeed, while free software can gain the upper hand in the short-term, a staff of industry-savvy *full-time* developers might raise the competition bar quickly. Ironically, because closed-source companies do not share the source of *their* neatest innovations, free software developers do not have the re-use advantage. Indeed, in my experience the present crop of free software developers do not reverse-engineer other products.

6 The term "clean room" came from the *legal* reverse-engineering of IBM's BIOS a few decades back. While the term is not in much use these days, the "educational use" clause in current U.S. Copyright laws confers a similar set of values. -Indeed, by ignoring the intellectual property patents of all environmentally and socially responsible nations, places like India have no such dilemma.

-Certainly not enough to enjoy much 'educational use': Few free software developers care about assembly language; Writing code from-scratch is simply too much fun."

Benefiting from open source projects is easy to understand. For example, while many love the 'freeness' of GCC, it is not the be-all and end-all of all C++ implementations. Other software developers can, and have, created superior C++ Tools.

The Common Code

"I'm glad we got THAT out of our system." smiled Debbie. "What C++ Compiler should we use then?"

"It shouldn't matter," I said. "When learning any open language, is not the compiler itself that we need to be most concerned about. It is the portability and support of their tools and **framework**."

"By frameworks, to we beginners you mean the software that others have written for us to use," said Debbie. "Make sure you tell your readers that."

"Well," David began "we have heard you talk a lot about your Open Source Project. Why not get started with that?"

"Good idea" Debbie added. "Along the way we can help you port it to other operating systems."

"Your help would be welcome" I added "Writing any interesting cross-platform framework is a lot for one person to do alone.

"Since Debbie is more interested in Mac OS X, and Dave is interested in Linux, I will leave the final compiler selection up to you. Later on we can spend a few nights getting our namespace working on our favorite development tools. Agreed?"

After nodding their heads vigorously, Dave added, "Both of us have Windows laptops. In order to get things jump started tonight, why not show us how to get things started by using GCC?"

Compiling a Program

To get my friends started using their Microsoft Windows laptops, we networked over to **cygwin.com**. After they downloaded and installed the free compiler set on their machines, we parked ourselves around my laptop. Then, upon opening up a command-line window, I typed in the following:

g++ hello2.cpp

“The **g++** may look strange, but is part of GCC.” is said. “Once we activate g++ to compile the program, a executable program is created.

On Windows the compiler creates a program named **a.exe**. Since we are in the same folder as what the compiler created for us, we can run that program by typing either 'a' or 'a.exe'."

Console Streams (cin, cout, & cerr)

After we ran the program I said "Notice how those dual greater-than and dual less-than operators allowed us to write "Hello world!" to the screen, as well as read a number from the keyboard:

```
int main(int argc, char *argv)
{
...
   cout << "Hello world!";
...
   cin >> iNum;
...
}
```

"Writing to **cout** is easy enough to understand. But notice how reading information from **cin** performed some magic: It automatically converted what we typed on the keyboard into something that we can use in our program. In this case, that variable is called `iNum`. iNum is something we can change.

"Both **cin** and **cout** will convert data into variables for us. When our programs encounter problems, there is also a place to write errors.

“Let me guess” said Debbie. We call it cerror?

“Almost!” I said chuckling. “It is known as ***c*err**.”

Dave added “The reason why having a place to write errors is a great idea is a relatively advanced concept[7]”

Agreeing, I added “Good point. -For the moment, we should merely note here that whenever our programs want to document an unexpected problem, that we should consider sending a message or three to *cerr*. rather than *cout*[8].”

---

7 Linux, OS X, and DOS/Windows allows programs to connect together via cin, cout, and cerr. While some notation is shared, other notation is used to do the exact same things in a different way. (i.e. Think “educational use”, here :)

8 For the impatient, feel free to Google the phrase “This Old Pipe” and “Nagy” for a complete description, as well as several C++ demonstration programs.

"By way of understating C++, when using streams notice how the logical direction of those mathematical signs - or operators - indicate whether data are flowing **in** or **out** of the variable. By giving us such a strong graphic hint at whether the data are coming or going, cout, cerr, and cin represent streams, or places where data flow to and / or between our variables."

```
cout << "Hello world!";
cin >> iNum;
```

"Some people refer to those places as data *sources*, or data *syncs*" Dave approved. Turning to Debbie, he explained "a data *source* is were data flows from. A data *sync* is where data flows to."

"It sounds a lot like kitchen plumbing to me" Debbie chuckled.

"Well" I noted "there is the concept of *piping* data between programs, but – as Dave says – it is an advanced concept. We will discuss *processes, pipes,* and *pipe-fittings* later!"

Looking furtively at Dave, afte rlightly knudging him in the ribs, Debbie whispered He's kidding us, right?" "Not even a little bit" Dave replied. "Piping data between processes & devices is what has made Unix such a excellent choice for serious research, diagnostic, and deployment use."

"Indeed" I added "From network connections to simple files, using streams allows many types of data to flow back and forth between our programs. Best of all, data can be translated from what a user types, into a more program-friendly format."

Intimidation Factor

"While all of the greater than and less than signs can be a little intimidating at first, after a while you get used to it." Dave noted.

"Indeed" I agreed. "One way to keep from being intimidated is to use comments:

```
/*
Every program begins with a main.
We will discuss how many ways to
use it in this book.
*/
int main(int argc, char *argv[])
{
   ...
}
```

"In C / C++, surrounding a block of lines with /* and */ allows us to leave bodies of notes in our code. For maximum efficiency, the pre-processor typically removes comment lines.

While large, or block comments are fine, in C++ prefixing any line with two forward slashes allows us to also quickly add short, single line comments.

```
...
int main(int argc, char *argv[])
{
   // How write something on the console -
   cout << "Hello world!";
   ...
}
```

"By using both block and single-line comments in C++ we can do a lot to demystify what we write."

A Classy Example

Dave said: “On any computer the file system is usually the most important place for streams. Not only is the file system where everything from the operating system to our programs and data stored, but these days the hierarchical file system - where directories contain folders and other files - is practically universal. I can remember when that was not the case.”

Debbie added: "Be sure to tell your readers that a directory and a folder are the same thing."

I agreed, then opened a Directory class from my own namespace. Because directory and file classes are missing from most frameworks, I began to point out some of the more interesting features of some of my own C++ creations.

After a few hours discussing the hundreds of files in the open source project, we went home for the night.

The next day, over lunch the next day, Dave said “Your Open Source namespace is huge. For your book, you should gradually copy things like your **Directory** and **File** classes into it. Show us how to add them to a new namespace; Allow developers to experiment with each a little at a time.”

Sage advice!

Day Two: Jumping Right In!

That night I was ready to tackle the project from Dave's point of view “In C++ like other object oriented languages” I began, “the **class** is where most learners like focus their attention. As the basic building block of all of re-usable examples, we can review the features of C++ as we examine the way I created one very popular class.

"Since Directories and Files are so commonplace, they are a good place to start when examining a C++ Framework.

```
#include <iostream>
#include "Joy.hpp"

using namespace Joy;

int main(int argc, char *argv[])
{
   Directory dir("c:\\windows");
   vector<File> fileList;
   if(dir.Query(fileList) == true)
      {
      cout << dir.Name();
      cout << " has " << fileList.size();
      cout << " files and folders.";
      }
   return 1;
}
```

"Wow. That looks nasty" Dave said. "I presume that the == sign is used to test if something is equal to something else?"

"Yes." I said. “In C++, when we want to assign something, we use a single equal sign. When we want to test if two object are equal, we use the == operator.” Denise added “In Java we use equals(). -Dave and I were just discussing how it works the same for .Net on our way over here tonight.”

“Operator overloading is something that lots of C++ Developers enjoy” I said. “Much like using pointers or a pre-processor however, in the hands of a novice it can get you into trouble. That is why C++ is seen as an expert's language. We can do a lot of extremely cool things. Things so efficient & scary, that the developers of other languages did not want you to even be able to think about. -But let's get back to the basics.”

Using If, Blocks, and Statements in C++

"Using an **if** statement is easy enough to understand. All we have to do is place a **test condition**, after the **if** keyword, in parentheses. If our test condition evaluates to **false**, then whatever follows the if statement will not run:

```
...
   // Demonstration of a single statement in a block
   if(dir.Query(fileList) == true)
      {
      cout << "Your Folder Listing";
      }
...
```

“In C++, matching pairs of curly braces are used to combine a list of statements into a single statement block, or block for short. In C/C++, we can alternatively end a simple, single-line **if** statement, with a semicolon:

```
   if(dir.Query(fileList) == true)
      cout << "Your Folder Listing";
```

"An interesting rule of C /C++ is that a block can be inserted anywhere a single semicolon-terminated statement can compile (and vice versa). So if we only wanted to execute a single statement, we could either put that single statement in a block by itself or choose to write without the statement block all together.

“Java and C# do not like us to do that.” agreed Dave and Deniese “It looks odd, but I'll bet you get used to it?”

“You do” I insisted. “-But let's get back to discussing how we use headers to find code.

Header Files

“By simply including the file that contains the Directory Class, we will copy all of the code that we need to use it. Because the pre-processor inserts everything into our file, C/C++ programmers try to limit the amount of borrowed source code they copy in.”

“Yes” added Dave “Most people like to include only what they want others to see in the header files. Everything else can be protected by a library.”

“Exactly” I agreed “Like skimming the outline out of a book, the file to be included is commonly referred to as a ‘***header file***.’ When we use **#include**, the name of the file to be copied into our program can be any file that the compiler can find.

“While not required, most C/C++ developers like to separate our definitions, from our implementations. While empty class definitions can look similar, in C/C++ we are allowed to have macros, constants, and functions defined outside of a class. -At least not at the moment.”

“What do you mean by *'at the moment*' ”? Asked Debbie.

“”Well” I said “Keep in mind the Java was created to make software development safe

for new software developers. Others companies make re-use so easy that people do not complain when *entire frameworks* are company-deprecated in favor of purchasing new ones. -Yet from templates to streams, just about everything in Java and C# has been borrowed from C/C++. Indeed, as yesterday's new software developers become more comfortable with advanced topics, we should not be too surprised to see the trend of borrowing advance ideas from C/C++ continue."

"Well" observed Debbie "I would not mind having a pre-processor of Java!"

"Exactly." agreed Dave. "Even more advances concepts – like pointers and templates - are why C++ is often seen an a software development language for experts. It is why books on C/C++ often need just as much discussion, as they do code."

"Excellent insight" I agreed. "In C/C++, the amount of code being included should be small. The rule of thumb is to #include only what is necessary to get a program to compile."

Include Path

Looking at Dave, I asked "Like the pre-processor, using #include in C++ works the exact same way as it does in C. Would you care to take the discussion form there?"

"That is nice to know" smiled Dave "All header files will either be located somewhere along a programmer-maintained set of folders. Most often referred to as the 'search path', the list of directories often includes the exact same folder as the file being compiled.

Environment Variable

Sometimes the search path will be specified using a special environment variable. Because environment variable are common to all programs on a computer, the compiler knows where we need it to look for the files to include."

"Excellent" I concurred. "No matter where the search path is defined, the purpose of the path is to tell the computer where to look for something to include. So in addition to the quotation marks, C/C++ Programmers can also place < and > around a file name when we want to find a file somewhere along the path. Just remember to use a use simple pair of quotation marks when that search path is to be ignored:"

```
#include <iostream>  // Header is somewhere on the search the path.
#include "Joy.hpp"   // Header is close to what is compiled.
...
```

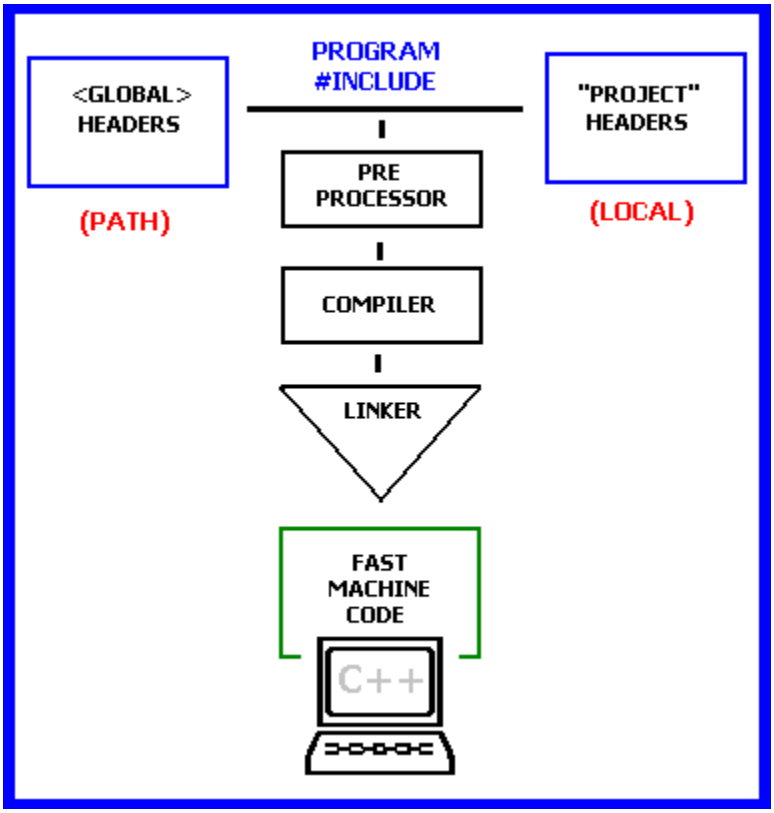
"What happens when there are two file along the path?" asked a new voice from the door. Startled, we look ed up to see yet another face from work. "Well" I said dryly "then the first one wins!"

“Hello Darrel” Debbie said “Glad to see you could make it.”

“The more the merrier” I said. “It is always nice to see a project manager outside of the office. Feel free to have a seat!”

Local & Global Include Paths

Opening up a common drawing tool, I said "Putting it all together, the relationship between local headers found next to your software, global headers shared on the path between all projects, and your program looks smoothing like this:



“I have always wonders what you C/C++ jocks mean when you talk about headers” Dave laughed.

“Yes” said Debbie. “… Java most often includes everything - not just class definitions.”

“In C++, our headers often contain only functions, pre-processor directives, and / or class definitions.” I said “Unlike in Java or C#, a single class can – and arguably should - be in different source-code files.

(After I noted that classes did not contain everything in C/C++, I will never be able to describe the look Darrel gave me – It looked a little like “Bill The Cat.”)

Selective Compilation

Undaunted, I continued: “Because the compiler does not have to re-compile everything, programs can be created quickly. Because the pre-processor inserts code into a single temporary file *before* that copy of our code is compiled, the overhead of looking into a seemingly endless set of Java *Packages*, .Net *Assemblies*, or even C/C++ *Libraries*, may be avoided.

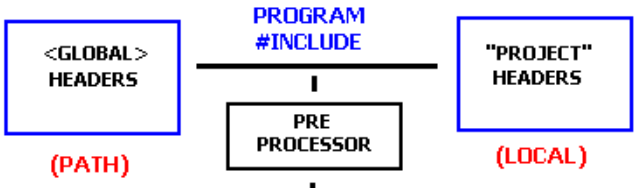
"So back to headers: In C/C+, having two ways to ask the pre-processor to lookup a header allows us to manage two ways to find code. To better understand why two sets of folder locations can be important, consider how we write software. While some things are reliable, others are often much less so. Hence when we see:

```
#include <iostream>
```

A developer is telling the pre-processor that the header is located within a GLOBAL set of folders. We can infer that the header is in used by a lot of people; Trusted.

On the other hand, when we see:

```
#include "Joy.hpp"
```



We can infer that for some reason this header needs to live closer to home. Located right next to the file that includes it, either the header contains intellectual property, or it is otherwise not ready to be shared.

The Linker

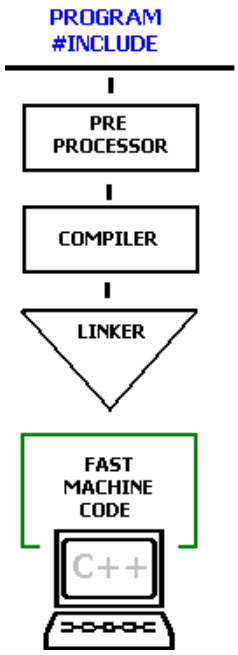
“But the real magic is the linker: Even if headers contain everything (the default Java or C#-Style classes), the pre-processor inserts *everything* we ask it to include into our temporary file. Undaunted by this *worst-case* scenario, C++ Programmers know that even when far too much code is being included, that a *linker* will make things smaller.”

“So the job of that linker is to include *only* the things that we need – not the entire namespace - into the final program.” said David.
“Yes” I replied. “Having a linker remove things you do not need is another reason why the overall hard-drive size of a C/C++ program is always a lot smaller that that of just about any other language.”

“Now you are talking about static linkage” said Dave. “Yea” added Debbie, glancing sideways at Darrel. “You might want to save that discussion for a more advanced section!”

“Gewd 'poi-nt!” Darrel agreed, trying his hand an Australian accent. “Lets keep it all topside tonight, mates.”

We laughed.



The Joy Namespace

"So take a look at that **using** and **#include** set once again," I said:

```
...
#include "Joy.hpp"
using namespace Joy;
...
```

"The pre-processor can include a lot of information. Unlike Java or C#, many classes and namespace might be in a single file. -That **using** statement is how we tell the compiler what we will be interested in **using** items from **Joy**.

Re-Factoring

Chuckling, Darrel said "Debbie told me that you were going to start a new project. Is it true that we are going to be copying form your open source project?"

"Yes" I said. "When I started the project, I was just learning C++ myself. Copying from one project into this one will give me an opportunity to review – and improve – the code. While some changes will undoubtedly be trivial, we might have a chance to do some re-factoring too."

Responding to Darrel's puzzled expression, Debbie added "Re-factoring is when we can change the way a class is used to make it more suitable for us to re-use."

"I knew *thaaaat* said Darrel, waving his hand in grand, circular, but comical way –A gesture that somehow assured us that he would be buying us lunch sometime soon.

"Smiling, I added "In short, only because this program included the "Joy.hpp" **header**, we can use the Joy **Namespace**. If we did not have the header available somewhere in *either* the **local** or **global** include path, the compiler would not be able to discover anything about the namespace. We would get an compile-time error[9].

Seeing that Darrel nodding his head at the above Diagram, paging back to the previous source code, we continued:

---

9 Modern software developers use software that *constantly* compiles your code. Because those tools integrate your software-editing chores into the software-development environment, such tooling is commonly referred to as an "Integrated Development Environments", or IDE, for short. For crafting C/C++, popular IDE tools include NetBeans, Eclipse, Visual Studio, and C++Builder ... but there are literally dozens of IDE choices.

```
#include <iostream>
#include "Joy.hpp"

using namespace Joy;

int main(int argc, char *argv[])
{
   Directory dir("c:\\windows");
   vector<File> fileList;
   if(dir.Query(fileList) == true)
      {
      cout << dir.Name();
      cout << " has " << fileList.size();
      cout << " files and folders.";
      }
   return 1;
}
```

“One of the things I put into the Joy Namespace last night was a Directory Class:

```
...
int main(int argc, char *argv[])
{
   Directory dir("c:\\windows");
...
}
```

By now it should be obvious that while the **Directory** name is found no where else, that *Directory* should probably be found in the Joy namespace?

Indeed, unlike Java and C#, but default we can name our files and include locations anything we want. In this case, the *Directory* class need not be in a file that has the same name. Neither does the *Joy* Namespace have to be in a folder that has it's name.

The freedom to organize our code any way want drives a lot of Java and .Net developers crazy.”

Classes are like Recipes

Denise added “When I was learning how to program, an instructor told me that *classes* are like recipes.

In that Directory example, once we have told the compiler that we want to use the ‘Directory’ recipe, it will know how to create a variable to use it in the real world. Once the compiler has created a variable, most people refer to the results as either an ‘Object,’ an 'instance of an Object', or simply as an 'Instance.'

“Once created by the compiler, Objects can actually be used.” she continued “That is why this example demonstrates how we can use the *Directory* class to create an entire set of *File* Objects.”

Introduction to Templates

"Thanks for sharing that, Denise!" I smiled "-Reviewing that set of *File* idea in our next example, we can see how **File** Objects are being created & collected together into a handy container. Known in C++ as a **vector**, witness how an entire *collection* of files are being created, then returned, by the Directory Object:

```
...
   vector<File> fileList;
   if(dir.Query(fileList) == true)
      {
      cout << dir.Name();
      cout << " has " << fileList.size();
      cout << " files and folders.";
      }
   return 1;
...
```

"Like everywhere else, an array in the standard C++ STL Framework is used to represent a set of items. In this case, a collection of **File**s."

"That works a lot like Java" Debbie observed. "But you need to tell your readers about all that junk on that vector definition line. That less-than and greater-than 'stuff threw me the first time I saw it!"

```
vector<File> fileList;
```

"Me too" I recalled "That code tells us that the vector class is actually is a reusable **template**" I said. "One that is being created to manage **File**. Afterward, we can use the **fileList** Object to do everything that **vector** will allow us to do.

Once created, **fileList** become a variable. Much like any other variable we usually want to change it, or have it do something for us.

Returning to the example, David said: "While a bit strange to look at, using templates seems easy enough. -All you need to do is to put the name of what you want to create between a less-than & greater than symbol[10].

More Template Discussion

10 At this point, the comment was that "You also do not have to use the ***new*** keyword." -While both Dave and Denise agreed, I said "Lets not get ahead of ourselves here. -Unlike other programming languages, C++ can indeed allow us to use ***new*** here – but for now, we are sticking to the basics: C++ has far too much to say about memory management than most beginners need to know!"

“It looks a lot like the code you use to read and write data using cout and cin.” David also observed:

```
...
   vector<File> fileList;
   if(dir.Query(fileList) == true)
      {
      cout << dir.Name();
      cout << " has " << fileList.size();
      cout << " files and folders.";
      }
   return 1;
...
```

“Yes” I agreed. “Keep in mind that templates look a little different what we use for reading from, or writing, to streams. -Rather than using the *dual* greater or lesser signs used by streams, the single-character-notation more resembles the way that we define a global *#include*. -But because we do not see that # sign anywhere, we know that are talking to the compiler, not the pre-processor.

```
...
   vector<File> fileList;
   if(dir.Query(fileList) == true)
      {
      cout << dir.Name();
      cout << " has " << fileList.size();
      cout << " files and folders.";
      }
   return 1;
...
```

“When the compiler is looking for a class for template use, all we have to do is to remember to put the name of the class we are looking for in-between the < and > characters.

A Few More Control Statements

“Once we have an array, we can use many of the same language statements to manage what we do with it. The most commonly used keywords in C++ include **while** and **for**.

Using While

Turning back to the keyboard, I updated the example to demonstrate using the **while** keyword:

```
...
   if(dir.Query(fileList) == true)
```

```
        {
        cout << "Listing of " << dir.Name() << endl;
        size_t ss = 0;
        while(ss < fileList.size())
           {
           File file = fileList.at(ss);
           cout << "... " << file.Name() << endl;
           ss++;
           }
        }
...
```

"While the while statement looks like **if**, it does the same job in C and C++ as everywhere else. The *while* keyword continues to execute a list of statements, until it's *test condition* is no longer true."

Wherefore size_t?

Debbie asked "What is that **size_t** thing all about?"

"All we need to know about size_t is that whenever something is stored into memory, size_t represents the number of bytes we will use to reach, or *index into* it. On Wintel, today size_t is the size of an unsigned integer. Size_t might be something else, somewhere else."

"That makes sense" Dave chimed in "In the days of 8-bit computing, everything was within an 8 bit reach. As processor and memory sizes increased to 16, 32, 64, and now to 128 bits, having something like a size_t defined allows software to continue to work even if memory management schemes need to do something odd in the meantime."

"Exactly" I said, typing in a short example:

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

int main(int argc, char *argv[])
{
   cout << "The sizeof(size_t) on this computer is ";
   cout << sizeof(size_t);
   return 1;
}
```

size_t

"The C standard says only that **size_t** is the type returned by the **sizeof()** operator. Part of the C++ standard, size_t must be some sort of *unsigned* integral type[11]. The actual type

11 While saying that a integer is signed or unsigned might sound intimidating, the name itself says pretty

used is free to vary from one implementation to another. On a wristwatch, the size might be 6, 8 or 16 bits. On most desktops, 32 bits is prominent at the moment.

Unlike desktops, it is not unusual for more powerful servers to typically have size_t values of 64 or 128 bits. In short, the definition of size_t can even be main-stream size, or some bizarre platform-specific type."

"Much like we need to work efficiently with advanced computers" Dave agreed. That size_t convention would come in real hand if we were designing some futuristic light-based artificial intelligence or a supercomputer. Anything that might have a larger-than-processor memory capability."

"Just so" I smiled "All we need to remember is that whenever we work with something in memory, the C/C++ Standard says we should be using a type called size_t."

Using For

“Okay” I began “As you both know” I said as I updated the example to use another control construct "new software developers need to see how using the **for** keyword is a little more complicated. Unlike **while**, the **for** keyword requires three independent statements. One to initialize the loop, one to test the precondition, and one to run whenever a pass thru the loop is completed.

```
...
   if(dir.Query(fileList) == true)
      {
      cout << "Listing of " << dir.Name() << endl;
      for(size_t ss = 0; ss < fileList.size(); ss++)
         {
         File file = fileList.at(ss);
         cout << "... " << file.Name() << endl;
         }
      }
...
```

“That looks too familiar" said Debbie “There are lots of books that cover things like the “**do**” and “**do/while**” statements for C. Rather than a boring retelling of what everyone else forces us to wade through, how about just listing the keywords in the back of the book, trust to everything from Amazon to Goggle, and get on with it?”

"Same here" said John "If your readers need more information on keywords, then they

---

much what the concept is all about By reserving a signel bit to ingiace when a integer type s negative or not, the number of bits that can be used is necessarily decreased by one. So if we are using an unsigned char (typically 8 bits), a *signed* char would leave only 7 bits left to represent data. Yet reserving a signed bit also allows for an entire range of bits to be displayed. Hence, rather than representing an *unsigned* range of between 0 and 255, the equidistant range of -127 to 127 can alternatively be represented. Hence either signed or unsigned - for char – as well as other integer types - the same number of possible values can still be represented.

can Google their keyboards off. --You got anything new?"

"Okay" I agreed, just to chase away the feeling of boredom that had been filling the room. "Then hang on your keyboards!"

After morning **UsingFor.cpp** over to **UsingIterators.cpp**, I said "Let me push the bounds of your tolerance by introducing you to some more of the Standard Template Library!"

The STL

"There is a standard *template* library?" asked Debbie.

"Sure is," said Dave "That is where that **vector** class came from."

"We have a Vector class in Java" said Debbie "It was improperly designed. We have been told not to use it any more. We are supposed to use ArrayList, instead."

Smiling, I started "Well, taking time to make a standard often boils-out those oversights early on. -While the occasional 'doh!' can indeed be heard echoing outside of a completed standardization process from time to time, mistakes do not happen as often as they do with Java or .Net.

"Many have noted that in companies - as well as committees - whenever we have a vested community, that one has a much better chance of avoiding those type of embarrassing mistakes[12]. In fact, when it comes to Templates, C++ had them before Java and .NET were even dreamed of."

Do Your Worst!

"Okay" said Denise "Creating templates stump most Java Developers. In fact, so far it looks like there can be some pretty cryptic-looking stuff going on in C++, as well. While you are doing a good job explaining it as we go along, even in Java, using Vector can look pretty strange."

Smiling, she continued: "I would be willing to bet that at this point that your readers might be dying to ask you -just how intense can C++ get?"

"Well" I mused "We have already gone as far and fast as ever any introductory book on a new language has ever dared to go before. -In fact, I would go so far as to say that if the

12 Mistakes like arbitrarily adding synchronization overhead to a utility class. (I move this elaboration to a footnote because it is trivia – We did not want some to worry if it sounds like gobdely-gook: At this point, only tenured developers – and Google – should care enough to know what 'adding synchronization overhead to a utility class', means!)

reader has understood most of what we have been talking about so far, then she or he certainly has what it takes to become an expert C/C++ Developer.

"Yet if I were to push the envelope a little farther, then I remember when I first learned a little more about the STL. One of the oddest things I ever recall seeing, was my first iterator.

"The reader will probably feel the exact same way. -So by way of a tough "bonus section" challenger, see how well you can decipher an iterator loop:

```
#include <iostream>
#include "Joy.hpp"

using namespace Joy;

int main(int argc, char *argv[]) {
    Directory dir("c:\\");
    vector<File> fileList;
    if (dir.Query(fileList) == true) {
        cout << "Listing of " << dir.Name() << endl;
        for (vector<File>::iterator It = fileList.begin();
                  It != fileList.end(); It++) {
            cout << "... " << It.base().Name() << endl;
        }
    }
    return 1;
}
```

"When we start scanning the above example, we should feel perhaps a bit proud about what we have learned so far. -Building upon the previous example, the only thing different is what we do with that **for** loop:

```
for (vector<File>::iterator It = fileList.begin();
          It != fileList.end(); It++) {
    cout << "... " << It.base().Name() << endl;
}
```

"When we focus-in on the above statement, we can better see three things: First, we told **fileList** that we needed to get his first element. Known as **begin()**, what is returned is a iterator to a **File**.

"Once we have the iterator, we can use it to keep on *iterating* through the **collection** until we arrive at the **end()** of the list.

"Like the Directory, File, Array, and string classes in my open source project, STL Classes have many more class members to explore."

"This is also a lot like Java" noted Debbie. "But Java cannot use operators like ++ or **!=** to work through a list:"

```
for (vector<File>::iterator It = fileList.begin();
        It != fileList.end(); It++) {
    cout << "... " << It.base().Name() << endl;
}
```

“Yea.” said Dave “Microsoft has been struggling with operator overloading, too.”

“It is not an easy concept” I agreed. “Another one of those advanced features, in the hands of a novice, much like using pointers and the pre-processor, operator overloading can get a fledgling software developer into trouble, too.

“Your homework should probably be to see how many things you can do with the classes we have used so far.”

Reading and Writing to Files

“Before we go home, let's do something interesting.” Dave said.

“Yea” said Debbie. “How about some kind of command-line tool. Something that reads or writes to a file? -A program that does something useful?”

“I have just the thing!” said I. “It is a simple logging utility; One that I wrote years ago. Something that even Skype even used for awhile. -Though simple, I use it to keep track of my notes and daily activities.”

“Sounds great!” said Dave. “Show us how it works.”

After rummaging around my hard drive again, I opened up a copy of a command-line main for EzLog:

```
#include <iostream>
#include "EzLog.hpp"
using namespace std;

int main(int argc, char *argv[]) {
    if (EzLog::Main(argc, argv, cout) == false)
        cerr << endl << "Error: Logging failure." << endl;
}
```

I said: “Here is a good example of using **main** to call a **STATIC** class entry point. Notice how we **#include** the header, then pass EzLog what was passed to main. ”

“More strange operators!” noted Dave. “-What's up with that **::** ?” added Debbie.

“That **::** is there to tell the compiler that **EzLog** has a STATIC member. In this case, a static MEMBER FUNCTION called **Main**.

“While neither Java not C# use '**.** ' rather than '**::**', the Java and .Net Languages also copied the *static* concept from C/C++. In general, static members can be used create an API[13]-style Function; A constant signature that we can use WITHOUT making an *instance* of EzLog.

“In fact, I was inspired by Java to include *two* Main()s in EzLog:

```
class EzLog {
   public:
     ...
       static bool Main(Array<StdString>& array, ostream& os);
       static bool Main(int argc, char *argv[], ostream& os);
};
```

“-In C/C++ as elsewhere, as long as the function name has different *parameters*, then we can use the name as many ways as we want. Everyone in the industry calls that Member Name, or just Name *Overloading*.

*“*But you might enjoy this: Because I designed EzLog to use a **stream**, we can also use the class under a GUI, as well as at the command line. -If we look at a stream as simple a note pad, then all we need do is to pass EzLog a stream to write to. EzLog can then write the exact same console result into a list-box, or a text field.”

“Cool!” said Debbie. “I would be great to be able to test a class from the command line, but also have it look great in a GUI.”

“That is one of the reasons why I wrote the EzLog Class the way I did. -Because testing is so important – even under a GUI - whenever I am writing a lot of code, I let my computer automatically build and test my tools each night. By detecting any problems as soon as they occur, I can get people to fix a problem while changes are fresh in their memory. This same design pattern will work in just about any modern framework... Java and .NET, too.

“While the EzLog class has a healthy amount of code, let's see how simple it is to open and write a string to the file. For this example however, we will use the Able1 Namespace.”

“I thought your Open Source Projects was *stdnoj*?” said Debbie “What is this *Able1* thing?”

“Able1 is an updated version of stdnoj” I replied. “After several revisions, I decided to give it a better name, as well as to include a version number as part of the official

---

13 An Application Programming Interface (API) can be best thought of as one or more simple functions. Conceptually predating the rise of Object Orientation, an API is most often used to support classic, as well as Object Oriented, programming languages. By way of example, the Standard C Library is often looked upon as an API supported by many popular Operating Systems. While Object-Oriented APIs do indeed exist, whenever objects need to be created to use them, most savvy software developers refer to such as a *Framework*, rather than an API.

moniker.

“Adding the number looks stupid” said Dave “But the name upgrade a definite improvement” he smiled.

“My thoughts exactly” I recalled “Yet by including a version number as part of the name, we can use several namespaces side-by-side. In the case of Able1, we can continue to upgrade – even make major changes – to an Able2, Able3, and so-on - without breaking other projects.

So we have seen how to use Elog in a STATIC, or API usage. Byt ever good developer ought to know that just because a class has an API doesn't mean that we cannot use it as an Object.

Such is the design of the EzLog:

```
#include <Able1.hpp>
#include <a1extra/StdLog/EzLog.hpp>

using namespace Able1;

/*
 * Using EzLog from the Able1 Namespace.
 *
 */
int main(int argc, char* argv[]) {
    EzLog logger;
    logger.WriteLog("Hello Logger!");
}
```

“Because EzLog will read as well as write log entries, here is how we can list everything that the file contains:

```
#include <iostream>
#include <Able1.hpp>
#include <a1extra/StdLog/EzLog.hpp>

using namespace Able1;

/*
 * Using EzLog from the Able1 Namespace.
 *
 */
int main(int argc, char* argv[]) {
    EzLog logger;
    logger.WriteLog("Hello Logger!");
    logger.ShowLog(std::cout);
}
```

“This example should tie what we have been discussing together nicely.”

“Except for one thing” said Debbie. “Java developers are not going to understand why C++ Developers do not need to use a '**new**' keyword. Is it possible to use *new* to create an Object in C++?”

“I'm glad you asked!” said I, looking at David nodding vigorously “Both Java and .NET software developers can indeed use the *new* keyword here. -The only difference is where the object is created.

“When using the `new` keyword with EzLog, one can either use:

```
EzLog logger = new EzLog;
```

or

```
EzLog logger = new EzLog();
```

-Either will do the exact same thing.

Day Three: Getting Excited

By the time of our next meeting, Debbie was getting excited. “This is not a s difficult as I thought it would be” she said.

“Same goes for me,” said Dave. “How goes to book?”

“So far so good. I’ll email you a copy of what we come up with after tonight.

“Do you feel like looking at some software designed to collect quotes, or recipes?”

Their answers were the exact opposite of what I anticipated; While Debbie want to see some quotes, Dave was keen on writing something to manage just about anything else.

“What say we do both? If we create a class that works the same way to create, read, and write what we are interested in, I can write a class that will let us manage *any* type of data. Here are the key operations your classes should have:”

[INSERT PURE VIRTUAL BASE CLASS CRUD SIGNATURE]

Both Debbie & David - knowing only too well where I was going with this example - smiled as we opened our laptops, and got to work.

Polymorphism Explained

While I was working on the class that would control Dave and Debbie’s objects, I

remembered how amazed I when I discovered how writing to support unknown implementations was even possible. While the idea occurred to many people while we were using C, the advent of C++ made the creation of such open-ended frameworks much more commonplace.

[INSERT A UML INHERITANCE DIAGRAM HERE]

In C++ the only thing required to support an unknown object is a well defined base structure or class. Much like definitions found in a header file, by defining the generic operations that you will call in advance any class that uses the definition (or *class signature*) will work within our framework.

Virtual, Pure Virtual, and Concrete Class Definitions

In C++, the only twist to this signature saga is if any implementation exists for our base class. Not only do we either we have an implementation or not, but we can choose to force people who want to base their class off of our definition by specifically saying so in their own class definition.

Here is an example of a plain old C++ Class definition:

[CLASS DEFINITION]

Because there is nothing special required to create and use this class, it is often called a **concrete class**

Next in the feature lineup are **virtual** classes:

[VIRTUAL DEFINITION]

While a **virtual class** definition also provides a signature, pretending the virtual keyword allows us to tell the world that the member function can be overridden with your own implementation. Virtual base structures and classes are what make frameworks possible. While anyone is free to override virtual functions, the keywords of public, protected, or private govern who use any class members.

Finally, for the brave soles who want developers to always provide or specify an implementation, C++ provides the notion of a **pure virtual base class**:

[PURE VIRTUAL]

Creating a pure virtual class is easy. By adding a “0” at the end of the member function definition, we are effectively telling anyone interested in using our class that they will *have to* provide their own implementation.

While other class and member re-use gametes are possible, concrete, virtual, and pure virtual classes are the type of member function definitions developers need to write frameworks.

A Reasonable Quote Implementation

Debbie was the first to complete the task. Her quote definition looked something like this:

[The DebQuote Class; AUTHOR, QUOTE, and SUBJECT]

A Reasonable Recipe Implementation

[The DaveRecipe Class: Open-ended Array<T> usage]

By choosing to use an array of string, Dave made his implementation a little more complex.

The Controller Class

The nice thing about writing frameworks like this is that no matter how complex your friends implementation is, you never have to worry about more than what it takes to use your own signature to control the class. Pointer or a reference to a class that inherits from your base class allows your friends class to be used interchangeably.

If you understand how using a pre-defined base class allows us to write a controller, then congratulations are in order: You have just mastered your first designed pattern. Known as the **Model / Controller Pattern**, or simply **Controller** or **Framework Design Pattern**, this convention is one that allows a team of developers to work together.

Testing Your Classes

While I was working on what was later to be called the OmniController class, Debbie and Dave we also writing small programs of their own. Designed as they were to test their own programs before handing them to over for me to use, Dave and Debbie's test programs are know as **unit tests**.

For my part, I was busy writing the code that I would need to unit test my controller as well. Moreover, because my code was also designed to stress test the entire *system* once their classes became part of my own, such overall test cases are know as **system tests**.

The Rewards of Testing

Because each of us spent some time writing test cases for our classes when it came time for us to put things together we had very little to do.

“What do you think of my unit tests?” said Debbie.

“Loved them.” I replied. "How about we add a Test member functions to each of our classes?"

“Bad idea” said Dave. “While I always hate to loose my test case, leaving diagnostic code in a program makes for slower execution and larger sizes. People can also exploit them. We should always remove our test cases from our software.”

“Here is where the pre-processor can help us out again.” I said. “By defining a **manifest constant** we can rely upon the pre-processor to exclude our test scaffoldings when the definition is not found.”

Testing Frameworks

Here is what the resulting base classes look like:

[SHOW HEADER WITH VIRTUAL BASE CLASS & #DEFINE DEBUG]

We can even define a main() test case. One that re-uses my system test:

[SHOW HOW #DEFINE CAN CHANGE TO USE MAN TEST CASES]

I should note here that while the pre-processor provides is a quick way to include or exclude test code, there are better ways to manage test cases. For example, we could create a **testing framework**. Such frameworks leverage a series of related and stand-alone classes designed to test our objects. Because the object snaps-in to our testing code, test cases can be easily removed.

Regression Testing

This ability to switch between run-time and test-time implementation is one great way to perform **regression testing**. The advantage of regression testing is that you can quickly test for variance – or regressions – in software capabilities. By leveraging streams and standardizing error message, you can use freely available streams-monitoring commands and other tools to alert you when critical errors occur.

The Final Solution

After explaining how streams and standard error messages work together, both Debbie and Dave were keen to see how such a regression-testing framework would work:

"I'd like to learn more about those free stream searching tools" said Dave.

"ID like to see how we can use the pre-processor to swap our testing code in and out" said Debbie.

"Fair enough!" I concluded. "Since the word 'free' came up, we will press on with a GCC tool set. Fair enough?"

While I watched both of them nodding their heads, I opened an MSDOS prompt and activated a nearby overhead projector.