

According to Hoyle...

ANSI/ISO C++ 2009: Changes Coming to the C++ Language

by Jonathan Hoyle

March 2007

Last month we explored some of the useful features of the Standard C and C++ Libraries. This month, we look forward to the changes ahead in the upcoming C++ Language and updates to the Standard Library.

The Evolution of C++09

After the first C++ specification was ratified in 1998, a deliberate 5 year period of silence was taken to allow compiler vendors to catch up with the standard, and for the committee to receive defect reports against the C++98 spec. At the end of this 5 year period, the ANSI Standards Committee released an updated ANSI/ISO specification containing bug fixes and wording improvements. These defects were documented in *Technical Corrigenda 1* in 2003. Changes were mostly minor, reiterations of things intended but were not properly spelled out. The biggest change coming out of it is the making explicit that memory allocated by the `std::vector` container must be contiguous. This updated version of C++ is referred to as *C++ + TGI*, or sometimes simply *C++03*.

Subsequent to that, committee members began by accepting proposals on various changes to the C++ language. This initiative was called *C++0x*, meaning that the expectation was to ratify a new version of the language sometime in 200x. As time has passed, it has become obvious that the new language cannot be ratified any earlier than 2009, and so recently the initiative has become named *C++09*.

A Technical Report on Library Extensions was initiated in 2004 and was completed in January 2005 (this report became known as *TR1*). This report recommended a number of extensions to the C++ Standard Library, many of which come from the Boost framework. By April 2006, the Standards Committee accepted all the recommendations of TR1 with the exception of certain high-level mathematics libraries (which were thought to be too difficult for some vendors to implement). GCC 4.0 already has a port of much of TR1, all under the `std::tr1` namespace, so *Xcode* users may begin using them now. *Metrowerks CodeWarrior* 9 & 10 also has a port of TR1. TR1 can be viewed at:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf>

[Note: A second Technical Report on Library Extensions was initiated in April 2005 for further research, but this report is not expected to conclude before C++09 is finalized.]

The Standards Committee intends to complete the C++09 document by the end of 2007. There are two meetings planned in 2007 to finalize all outstanding issues: April 2007 in Oxford, UK and October 2007 in Kona, Hawaii. Assuming no other delays, a completed document for general review should be available in 2008, and so ratification can take place sometime in 2009.

Philosophy of C++09

With the ratification of C++98 being less than 10 years ago, the Standards Committee was not interested in any large, sweeping changes in the language. Instead, it was interested primarily in changes which make the language easier to use and more accessible to beginning programmers. One of the complaints that many people have against C++, in comparison to other languages, is that it is too *expert-friendly*. Many programmers are not interested in being experts at a programming language; rather, they wish to be experts in their fields and simply *use* C++. Although a few new powerful features are being added to the language, simplifying to a reasonable level is a goal.

Another goal was to err on the side of updating the Standard Library before changing the language core. Changing the language itself is inherently more risky and leads to greater backward compatibility problems. Library enhancements allow for greater flexibility with less risk. Consider automatic memory garbage collection for example: modifying the core language for self-cleanup (as Java and C# do) would involve much greater change in the language and necessary lack of compatibility. However, supplying a smart pointer class into the Standard Library would give the user these same capabilities without loss of backward compatibility.

Finally, the committee strived for improving real-world performance whenever possible. One of C++'s strengths is its performance relative to newer languages like C# and Java. The user base is aware of this fact, and many have made C++ their programming language of choice for this very reason. In 2003, IDC reported that there are over 3 million full-time C++ developers; it makes sense to improve the language for their usability and not try to turn it into something it's not.

The Cautionary Tale of EC++

In 1999, a consortium of Japanese embedded systems tool developers (including NEC, Hitachi, Fujitsu and Toshiba) put together a proposal for a language subset of C++. This subset would essentially be C++ with a number of language features removed which (they thought) were too complicated and could hurt performance. The features targeted for removal included: multiple inheritance, templates, exceptions, RTTI, the new cast operators and namespaces. This new language subset would be called *Embedded C++*, or *EC++* for short.

To the surprise of the consortium members, the EC++ compilers were not only no faster than their C++ cousins, but in some domains EC++ was actually slower! C++ founder Bjarne Stroustrup later explained that templates were used in much of the Standard Library to improve performance, and their removal put EC++ at a disadvantage. Upon hearing this, the dismayed members of the EC++ consortium scrambled to put together a new proposal: *Extended EC++*, which was simply EC++ with templates put back in.

When the Extended EC++ compilers became available, they were once again put to the test against their C++ cousins. To the consortium's bewilderment, once again the performance gains relative to C++ turned out to be negligible. Part of the problem was the consortium's ignorance of C++'s *Zero Overhead Principle*: "what you don't use, you don't pay for". After this final embarrassment, ISO refused endorse any of the EC++ proposals.

In 2004, inspired by the EC++ debacle, the C++0x committee called for a Performance TR to determine which features of the C++ language truly had the greatest penalties in performance. As it turned out, there were only three areas where there were any measurable performance issues:

- 1) free store (**new** and **delete**)
- 2) RTTI (**dynamic_cast** and **typeid()**)
- 3) exceptions (**throw** and **catch**)

Memory allocation and deallocation turned out to have the largest impact on performance; however, it is unlikely that you would want to use a language that did not allocate memory from the heap. As for RTTI and exception handling, many compilers have switches allowing you to disable these if necessary. Many modern compilers have greatly optimized their implementation of Exception Handling, making RTTI the only outlier. In any case, with the Zero Overhead Principle in place, simply not using a C++ language feature is no different than having it removed.

As for EC++, Stroustrup is quoted as saying, "*To the best of my knowledge EC++ is dead, and if it isn't it ought to be.*" To view the Performance TR, go to:

<http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf>

Embarrassments, Fixes & Improvements

Although the C++ standard in 1998 was an astounding achievement, there were a small number of flaws that remained. Some of these were simply oversights; others were known, but there had not been sufficient agreement for resolution. Bjarne Stroustrup described some of these as *embarrassments*, particularly when trying to explain them to novices. Some of these improvements include:

```
vector<vector<int>>> x;           // Finally, legal!
vector<double> x = { 1.2, 2.3, 3.4 }; // Initializing STL containers
stronger typing of enum's      // Enumerated types remain in their scope
extern-ing of template's       // No duplication across translation units
```

If you are not familiar as to how any of the above caused errors, you needn't even bother to understand why. They are problems that are going away in C++09. I will delve into the first item only (Stroustrup's biggest embarrassment), to give you a flavor of the problem. The flaw lies simply with the fact that C++98 parses the ">>" portion of `vector<vector<int>>> x;` as a right shift operator and generates an error; C++09 fixes this. One of the reasons this took so long is that ANSI/ISO committee members are very hesitant to put in *silent changes* in the specification. A *silent change* is one that would keep the meaning of some C++ code with generating an error. Surprisingly, the reinterpretation of ">>" within templates can yield a silent change, as this example shows:

```
template<int I>
struct X
{
    static int const x = 2;
}

template<>
struct X<0>
{
    typedef int x;
}

template<typename T>
struct Y
{
    static int const x = 3;
}

static int const x = 4;

cout << (Y<X<1>>>::x>::x>::x) << endl; // C++98 prints "3"
// C++09 prints "0"
```

ANSI/ISO C99 Synchronization

The ANSI/ISO C specification was updated in 1999 with a number of improvements in the language. Many of these improvements were simply acquiring behavior that was already legal in C++ but seemed to make sense

for C as well. Other changes were not part of C++, but the ANSI/ISO C++ committee in turn saw some of these features as valuable and are rolling these into the C++09 specification. These include:

```
__func__ // returns the name of the function within which it resides
long long // extended integral type, typically used for 64-bit integers
int16_t, int32_t, intptr_t, etc. // specific integer types as defined in <stdint.h>
Hex floating point types, eg: double x = 0x1.F0;
Complex versions of some math functions, such as arcsin(), arccos(), fabs(), etc.
Variadic macros, that is macros taking a variable number of arguments, such as:
#define S(...) sum(__VA_ARGS__)
```

For more information on *C99 Synchronization with C++09*, visit:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1653.htm>

Standard C++ Library Enhancements

The Standard C++ Library, including STL (the Standard Template Library), is a generous supply of useful containers and utilities. Despite its fullness of capabilities, there were still a number of components missing. C++09 fills these gaps:

```
regex: a long awaited regular expressions class
array<>: a 1-dimensional array class containing its size (can be size 0)
STL hash classes: unordered_set<>, unordered_map<>, etc.
(These do the same thing as their ordered counterparts, except using a hash table)
tuple<>: a templated tuple class of multiple types: tuple<int,int> x; tuple<double,void *,A,B> z;
```

Mac users are fortunate in that they do not have to wait until 2009 for the Standard Library changes: they are available today in gcc 4 (the compiler inside *Xcode 2.x*). These library additions are within the library namespace **std::tr1::** (“tr1” stands for *Technical Report #1*, the standard’s committee report defining these new classes).

For more information on *Standard C++ Library Enhancements*, visit:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf>

Thread Enhancements

Mac OS X programmers who have had to write Unix level multithreaded code will heartily embrace the new thread-capable features of C++09.

```
Thread Local Storage:
thread int x = 1; // global within the thread

Atomic Operations:
atomic
{
    ... // pauses other threads during scope
}
```

Parallel Execution:

```
active
{
  { ... }           // first parallel block
  { ... }           // second parallel block
  { ... }           // third parallel block
}
```

In the case of Parallel Execution, it is likely that **active** will be implemented similarly to the **register** and **inline** keywords in that it will be a compiler request only. If the compiler find that the overhead of creating parallel blocks in some given instance outweighs the savings, it will be free to ignore the request and run the blocks serially.

Clearly, these language features greatly simplify coding which would otherwise require the use of **pthread**'s, **mutexes** and a number of other objects. It should be pointed out that the aforementioned features are still being debated by C++09 committee members, so there may be some changes to what I have described prior to ratification.

For more information on *Thread Enhancements*, visit:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1834.html>

Variadic Templates

For years, the C language has allowed functions to have a variable number of parameters. Unfortunately, this was not true of template arguments within C++98. In C++09, templates can have a variable number of types. Here is an example in which a templated **DebugMessage()** function can take advantage of variadic templates:

```
// Prints to stderr only when DEBUG flag set
template<typename... TypeArgs>
void DebugMessage(TypeArgs... args)
{
  #ifdef DEBUG
    ...           // Implement writing to stderr
  #else
    // Do nothing if the DEBUG switch is off
  #endif
}

// Later in code
DebugMessage("The value of n = ", n);
DebugMessage("x = ", x, ", y = ", y, "z = ", z);
DebugMessage("TRACE:",
  " time = ", clock(),
  " filename = ", __FILE__,
  " line number = ", __LINE__,
  " inside function: ", __func__);
```

For more information on *Variadic Templates*, visit:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2080.pdf>

Delegating Constructors

Other languages, such as C#, allow one class constructor to invoke another. In C++98, this was not possible, thus requiring the class designer to create a separate initialization function if it wished to use common code across multiple constructors. In C++09, this becomes available, as the following example shows:

```
class X
{
    public:
        X();           // default constructor
        X(void *ptr); // takes a pointer
        X(int value); // takes an int
};

X::X(): X(NULL)      // calls X(void *)
{
    ...              // other code
}

X::X(void *ptr): X(0) // calls X(int)
{
    ...              // other code
}

X::X(int value)      // does not delegate
{
    ...              // other code
}
```

For more information on *Delegating Constructors*, visit:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1986.pdf>

Null Pointers

In ANSI C, **NULL** is defined as **(void *)0**. In C++, the use of **NULL** is deprecated. Why? Because unlike in C, it is illegal in C++ to assign a **void** pointer to any other type of pointer:

```
void *vPtr = NULL; // legal C, legal C++
int *iPtr = NULL; // legal C, illegal C++
// Cannot assign a void * to int * in C++!
int *iPtr = 0;    // legal C++
```

However, the proliferation of **NULL** in C++ code remains so great, many compilers simply generate a warning, not an error, when such a pointer assignment mismatch takes place. Others redefine **NULL** in C++ as **0**. Despite these occasional compiler courtesies, it is still very confusing for beginning C++ programmers, especially in examples such as these:

```
void foo(int); // Takes an int
```

```

void foo(char *);           // Takes a char pointer

foo(0);                      // Is this supposed to be a ptr or the number 0?
foo(NULL);                   // No matching prototype

```

For this reason, C++09 introduces **nullptr**, a type-safe null pointer which can be used with any pointer, but is not compatible with any integral type:

```

char *cPtr1 = nullptr;    // a null C++ pointer
char *cPtr2 = 0;           // legal, but deprecated
int n = nullptr;          // illegal
X *xPtr = nullptr;        // can be used with any ptr type

void foo(int);             // Takes an int
void foo(char *);         // Takes a char *

foo(0);                    // Calls foo(int)
foo(nullptr);              // Calls foo(char *)

```

For more information on *nullptr*, visit:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1601.pdf>

The Amazing Return of auto

When the C language first evolved, the **auto** keyword was used to specify to the compiler that a variable was being allocated on the stack, for example:

```

auto x;                    /* a variable named x, implicitly an int, is placed on the stack */

```

When ANSI C was ratified in 1989, the implicit **int** rule was removed:

```

auto x;                    /* illegal in ANSI C */
int x;                      /* OK, auto assumed */
auto int x;                 /* OK, but redundant */

```

Since that time, **auto** remained a keyword in the C (and later C++) languages, even though virtually no one had used it since the 1970's. After over 30 years of disuse, the C++09 standard introduces the **auto** keyword to mean the variable type is implied by the initializer:

```

auto x = 10;                // x is an int
auto y = 10.0;              // y is a double
auto z = 10LL;              // z is a long long
const auto *p = &y;         // p is a const double *

```

The savings becomes more significant with complicated types, such as the following example:

```

void *foo(const int doubleArray[64][16]);
auto myFcnPtr = foo;        // myFcnPtr is of type "void *(const int(*)[16])"

```

In addition, **auto** becomes useful for temporary variables whose types aren't important but merely just have to match. Consider the following function which walks through an STL container:

```
void foo(vector<MySpace::MyClass *> x)
{
    for (auto ptr = x.begin(); ptr != x.end(); ptr++)
    {
        ... // Code modifying the data
    }
}
```

Without **auto**, the type for variable **ptr** would be **vector<MySpace::MyClass *>::iterator**. Moreover, any change to this container, such as changing it from a **vector<>** to a **list<>**, or changing the class name or namespace, would require changes in the **ptr** variable definition, despite the fact its type is completely unnecessary to note (other than for the compiler).

Note that an initializer is still required to use **auto** for C++09:

```
auto x; // still illegal in C++09
```

But suppose you knew what type you wanted (based upon another variable) but did not want to initialize? The new **decltype** keyword is available for just such purposes, as the following example shows:

```
bool SelectionSort(double data[256], double tolerance);
bool BubbleSort(double data[256], double tolerance);
bool QuikSort(double data[256], double tolerance);

decltype(SelectionSort) mySortFcn;

if (bUseSelectionSort) mySortFcn = SelectionSort;
else if (bUseBubbleSort) mySortFcn = BubbleSort;
else mySortFcn = QuikSort;
```

For more information on *auto* & *decltype*, visit:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1984.pdf>

Smart Pointers

Smart Pointers are objects pointing to memory which are smart enough to know when to delete themselves, rather than rely upon the user to manage its deallocation. Virtually all modern languages, such as Java and C#, manage memory in this fashion and thus avoid memory leakages and overstepping. The C++98 Standard Library came a minimally smart pointer object, **auto_ptr<>**. Unfortunately, **auto_ptr<>** had some severe limitations to it, the most notable of which being that it used an *exclusive ownership* model. That is, the last **auto_ptr<>** receiving the assignment was the sole owner of the memory:

```
auto_ptr<int> ptr1(new int[1024]); // ptr1 has exclusive access
auto_ptr<int> ptr2 = ptr1; // ptr2 has exclusive access, ptr1 no longer
```

This is counter-intuitive, as one does not expect the source object to change in such an assignment. The C++ community has by and large rejected **auto_ptr<>** and its use is now rather minimal.

C++09 Standard Library introduced a smarter pointer object, `shared_ptr<>`. Its main difference over `auto_ptr<>` is that it uses a *shared ownership* model using *reference counting* to determine when the memory should be deallocated. For example:

```
main()
{
    shared_ptr<int> ptr1;                // null smart ptr
    ...
    {
        shared_ptr<int> ptr2(new int[1024]);
        ptr1 = ptr2;                    // both ptr1 & ptr2 own it
    }
    // ptr2 destructed, only ptr1 owns it
    // memory not yet deallocated
}
// ptr1 destructed, now delete is called on it
```

A `shared_ptr<>` can be treated as a pointer, so it can be dereferenced like `*ptr1` or call call methods upon the underlying data such as `ptr1->foo()`. The following are some constructors for `shared_ptr<>` that make it useful to use:

```
explicit shared_ptr<T>(T *ptr);        // Attaching to memory
shared_ptr<T>(T *ptr, Fcn delFcn);     // Attaching to memory and a user- defined deletion fcn
shared_ptr<T>(shared_ptr<T> ptr);     // Copy constructor
shared_ptr<T>(auto_ptr<T> ptr);       // Converting from an auto_ptr<>
```

Note this last constructor converting the data from an `auto_ptr<>` to a `shared_ptr<>`, making it easier for you to transition your previous code. There are some additional utilities made available as well, such as a `swap()` routine and two cast routines: `static_pointer_cast()` and `dynamic_pointer_cast()`.

Fortunately for Mac programmers, `shared_ptr<>` is part of the `std::tr1::` namespace, and thus is already available to Mac users using *Xcode 2.x* or higher.

For more information on `shared_ptr`, visit:

<http://www.informit.com/guides/content.asp?g=cplusplus&seqNum=239&rl=1>

Rvalue References

In C, function parameters are always passed by value; that is, a copy of the parameter always is passed, never the actual parameter. To modify a variable in a C, the function must pass the parameter's pointer, and then dereference the pointer internally to modify the data:

```
void foo(int valueParameter, int *pointerParameter)
{
    ++valueParameter; // parameter passed by value, so modifications are local to this copy
    ++pointerParameter; // pointer passed by value, so modifications are local to this copy
    ++*pointerParameter; // dereferencing pointer, so modifications are permanent
}
```

One of the powerful new features that C++ introduced over C was that of a reference, using the `&` operator. Functions in C++ could then have parameters passed by reference, thus allowing its data to be modified directly without the need of a pointer:

```
void foo(int valueParameter, int &referenceParameter)
{
    ++valueParameter;           // passed by value, so modifications are local to this copy
    ++referenceParameter;       // passed by reference, so modifications are permanent
}
```

References must be to lvalues, that is variables which can be modified. Rvalues, read-only or temporary memory, cannot be used:

```
int    myIntA = 10;
int    myIntB = 20;

foo(myIntA, myIntB);           // myIntA stays at 10, myIntB becomes 21
foo(1, myIntA);                // 1 passed in by value, myIntA becomes 11
foo(myIntA, 1);                // Error: 1 is an rvalue and cannot be passed in
foo(0, myIntB + 1);           // Error: myIntB+1 is an rvalue and cannot be passed in
```

Occasionally, it is useful to pass a parameter by reference even when there is no desire to modify its contents. This is particularly true when a large class or struct is being passed to the function, and you wish to avoid creating a copy of the large object:

```
void foo(BigClass valueParameter, const BigClass &constRefParameter)
{
    ++valueParameter;           // passed by value, so modifications are temporary
    ++constRefParameter;       // compiler error, cannot modify a const parameter
}
```

In C++09, a new type of reference is defined, that of an *rvalue reference* (the familiar type of reference from C++98 is now referred to as an *lvalue reference*). Rvalue references can bind to temporary data but act on it directly without the need of a copy. The `&&` operator indicates that a reference is an rvalue reference:

```
void foo(int valueParameter, int &lvalRefParameter, int &&rvalRefParameter)
{
    ++valueParameter;           // parameter passed by value, so modifications are local to this copy
    ++lvalRefParameter;         // lvalue reference makes changes permanent
    ++rvalRefParameter;         // rvalue reference makes changes local without a copy
}

foo(0, myIntA, myIntB + 1); // The temporary value myIntB + 1 is not copied but moved as is
```

One of the chief benefits of rvalue references is the ability to take advantage *Move Semantics*, that is, moving data from variable to variable without copying. A class can define a Move Constructor instead of, or in addition to, a Copy Constructor as so:

```
// Class definition
class X
{
```

```

public:
    X();           // Default Constructor
    X(const X &x); // Copy Constructor (lvalue ref)
    X(X &&x);      // Move Constructor (rvalue ref)
};

// Utility function returning X
X bar();

X x1;           // Default construction of x1
X x2(x1);       // x2 created as a copy of x1
X x3(bar());    // bar() returns a temporary X, memory moved directly into x3

```

The primary motivation behind Move Semantics is improving performance. As an example, let us suppose you have two vectors of strings which you would like to swap data between. Using standard Copy Semantics, an implementation might look like this:

```

void SwapData(vector<string> &v1, vector<string> &v2)
{
    vector<string> temp = v1; // A new copy of v1
    v1 = v2;                 // A new copy of v2
    v2 = temp;               // A new copy of temp
};

```

Using Move Semantics, you can bypass all of that copying:

```

void SwapData(vector<string> &v1, vector<string> &v2)
{
    vector<string> temp = (vector<string> &&) v1; // temp now points to same data as v1
    v1 = (vector<string> &&) v2;                 // v1 now points to same data as v2
    v2 = (vector<string> &&) temp;               // v2 now points to same data as temp
};
// No copies are made, only pointers are exchanged!

```

For more information on *Rvalue References*, visit:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1690.html>

Concepts

Concepts are constraints on types, useful for templated classes and functions. Take for example the definition of `std::min<>()`:

```

template<typename T>
const T &min(const T &x, const T &y)
{ return (x < y) ? x : y; }

```

This definition for `min<>()` allows for any type to be passed into it, despite the fact that it makes sense only when the templated type has the `<` operator defined upon it. Modifying this definition so that it takes a concept instead of a generic type solves this problem, and thus allows `min<>()` to be defined differently on another set of types

that does not define the < operator. First, we define the concept:

```
auto concept LessThanComparable<typename T>
{
    bool operator<(T, T); // We require the < operator be defined
};
```

With our concept now defined, we can modify the definition of `std::min<>()` to use a concept instead of a type, as follows:

```
template<LessThanComparable T>
const T &min(const T &x, const T &y)
{ return (x < y) ? x : y; }
```

For more information on *Concepts*, visit:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1758.pdf>

Other Additions in C++09

In addition to the features already described, here is a small list of some other sundry additions being added to the C++09 language:

New char types: `char16_t`, `char32_t`:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2018.html>

Static asserts (from `Boost::`):

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1720.html>

Template Aliasing:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1489.pdf>

Overloading operator `.`(`operator .()`):

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1671.pdf>

Type Traits: `is_pointer()`, `is_same()`: http://home.twcny.rr.com/hinnant/cpp_extensions/builtin_traits.html

New `for` loop (a la `foreach`):

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2049.pdf>

Extern Templates:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1987.htm>

New Random Number Generator:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2079.pdf>

Advanced/Active Topics

There are a number of other features that the C++ Standards committee is still debating. Although it is likely that at least one or two of these will make it into the final standard, unfortunately, time is simply not permitting to allow all of them in. Here is a partial list of some topics which may, or may not, become part of the C++09 standard:

Transparent Garbage Collection:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1833.pdf>

Dynamic Library Support:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1496.html>

Memory Alignment Facilities:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1971.pdf>

Explicit Conversion Operators:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1592.pdf>

Extended Friend Declarations:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1791.pdf>

Explicit Namespaces:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1691.html>

Extensible Literals:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1892.pdf>

Not Available for C++09

There are many very cool features that were discussed by the ISO committee that just simply lack sufficient consensus or priority to be available in C++09. For example, one of the most common requests was the creation of a standard cross-platform GUI API, but such a thing was way out of scope for anything the standards committee would formalize. Others, such as Modules, seemed very much in reach but was not as important as other features. Here is a partial list of items that (unless something miraculous happens) will not be available in C++09:

Infinite Precision Arithmetic:

<http://en.wikipedia.org/wiki/Bignum>

Properties & Events:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1615.pdf>

Contract Programming:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1962.html>

Exclusive Inheritance:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1492.pdf>

Decimal Library:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1977.html>

A Full Multithreaded API:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2094.html>

Modules:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2074.pdf>

The C++0x Standard Library Wish List is up to revision 6 and can be found at: http://docs.google.com/View.aspx?docid=ajfb44js8vjx_bchdmtqvpxv4

Further Reading

The actual proposals for C++09 are available to read from the C++ Standards Committee web site at <http://www.open-std.org/jtc1/sc22/wg21/>. In addition, the following are links to various articles discussing C++09:

C++09: A Glimpse into the Future:

<http://www.devsource.com/article2/0,1895,2061094,00.asp>

C++09: The Road Ahead:

<http://www.informit.com/guides/content.asp?g=cplusplus&seqNum=320&rl=1>

C++09: Proposals by Statuses:

<http://www.informit.com/guides/content.asp?g=cplusplus&seqNum=323&rl=1>

C++: Predictions for 2007:

<http://www.informit.com/guides/content.asp?g=cplusplus&seqNum=321&rl=1>

State of C++ Evolution:

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2122.htm>

Toward a Standard C++0x Library:

<http://www.gotw.ca/publications/mill20.htm>

A Brief Look at C++0x:

<http://www.artima.com/cppsource/cpp0x.html>

C++0x: The New Face of Standard C++:

<http://www.informit.com/guides/content.asp?g=cplusplus&seqNum=216&rl=1>

The Design of C++0x:

<http://www.research.att.com/~bs/rules.pdf>

C++0x: Wikipedia Entry:

<http://en.wikipedia.org/wiki/C++0x>

C++ in 2005:

http://www.awprofessional.com/content/images/art_stroustrup_2005/elementLinks/DnE2005.pdf

Conclusion

Many of the changes in the next version of C++ are available to programmers today, particularly those that are a part of the Standard Library. Even of those not yet available, C++ developers ought to prepare themselves for the new language features. With increased readability and comprehension, C++ appears to have a very exciting future, even still.