

What is C++0x?

Bjarne Stroustrup

Abstract

This paper illustrates the power of C++ through some simple examples of C++0x code presented in the context of their role in C++. My aim is to give an idea of the breath of the facilities and an understanding of the aims of C++, rather than offering an in-depth understanding of any individual feature. The list of language features and standard library facilities described is too long mention here, but a major theme is the role of features as building blocks for elegant and efficient software, especially software infrastructure components. The emphasis is on C++'s facilities for building lightweight abstractions.

Introduction

What is C++0x? I don't mean "How does C++0x differ from C++98?" I mean what kind of language is C++0x? What kinds of programming are C++0x good for? In fact, I could drop the "0x" right here and attack the fundamental question directly: What is C++? Or, if you feel pedantic, "What will C++ be once we have the facilities offered by the upcoming standard?" This is *not* an innocent "just philosophical" question. Consider how many programmers have been harmed by believing that "C++ is an object-oriented language" and "religiously" built all code into huge class hierarchies, missing out on simpler, more modular approaches and on generic programming. Of course C++ *also* supports OOP, but a simple label – especially if supported by doctrinaire teaching – can do harm by overly narrowing a programmer's view of what can be considered reasonable code.

I will discuss the "what is?" question in the context of C++0x, the upcoming revision of the ISO C++ standard. The examples will emphasize what's new in C++0x, but the "commentary" will emphasize their place in the general C++ design philosophy. Despite "x" in "C++0x" becoming hexadecimal, I'll use the present tense because most features described are already available for exploratory or experimental use (if not necessarily all in the same implementation). Also, for the benefit of people with an allergy to philosophy, I will discuss ideas and facilities in the context of concrete code examples.

The rest of this paper is organized like this:

- Simplifying simple tasks
- Initialization
- Support for low-level programming
- Tools for writing classes
- Concurrency
- The sum of the language extensions

- Standard library components
- So, what does this add up to?

That is, I roughly proceed from the simple to the complex. If you want more examples, more details, and/or more precision, see my online C++0x FAQ [Stroustrup], which contains references to the actual proposals with the names of the people who did most of the work and to the draft standard itself [Becker, 2009].

Simplifying simple tasks

Much of what there is to like and dislike about a programming language are minor details; the kind of minor design decisions, omissions, and mistakes that escape academic attention. For C++0x, quite a bit of attention has been paid to “removing embarrassments” found in C++98 and simplifying and generalizing rules for the use of the language and standard library.

Deducing a type, etc.

Consider:

```
cout << sqrt(2);
```

Obviously, this should work, but it does not in C++98: The compiler couldn't decide which of the several overloads of `sqrt()` to pick. Now it can (there are now sufficient overloads to get a right answer), but what if you want the value “right here” and not as an argument to an overloaded function or a template?

```
auto x = sqrt(2);
```

By using `auto` instead of a specific type, you tell the compiler that the type of `x` is the type of its initializer. In this case, that happens to be `double`. In many templates, `auto` can save you from over-specification and in some cases `auto` saves you from writing some long-winded type:

```
for (auto p = v.begin(); p!=v.end(); ++p) cout <<*p;
```

Say that `v` was a `vector<list<double,Myallocator<double>>>` and that a suitable definition of `operator<<()` for `lists` was available. In this case, I saved myself from writing something like

```
for (vector<list<double,Myallocator<double>>>::iterator p = v.begin();
     p!=v.end();
     ++p)
    cout <<*p;
```

I consider that improvement significant – even more so for a reader and a maintainer than for the original programmer.

“`auto`” is the oldest improvement in C++0x: I first implemented it in C with Classes in the winter of 1983/84, but was forced to take it out because of the obvious(?) C incompatibility. Please also note that I did not say `vector< list< double, Myallocator<double> > >`. We can now do without those annoying extra spaces.

Such “trivial improvements” can matter disproportionately. A tiny stone in a boot can spoil a whole day and a tiny irregularity in a language can make a programmer’s day miserable. However, a programmer cannot simply remove a language irregularity – at best it can be hidden behind an interface until a standards committee gets around to the problem.

Range for loop

Let’s step up one level from the minute to the small. Why did we play around with iterators to print the elements in that vector? We need the generality offered by iterators for many important loops, but in most cases, we don’t care about details, we just want to do something to each element, so we can say:

```
for (auto x : v) cout << x;
```

That’s not just shorter, but also a more precise specification of what we do. If we wanted to access neighboring elements, iterate backwards, or whatever, we can do so (using the familiar and general `for` loop), but with this “range for loop” we know that nothing complicated happens in the loop body as soon as we see the header of the `for`-statement.

This range-`for` loop works for all ranges; that is, for every data structure that has a beginning and an end so that you can iterate through it in the conventional way. So it works for `std::vector`, `std::list`, `std::array`, built-in arrays, etc.

Note the use of `auto` in that loop; I did not have to mention `v`’s type. That’s good because in generic programs that type can be very hard to express. Consider:

```
template<class C> print_all(const C& v)  
{  
    for (const auto& x : v) cout << x;  
}
```

This can be called with any “container” or “range” providing a `begin()` and an `end()`. Since `C` was declared `const` and I don’t know the size of the elements, I decorated `auto` with `const` and `&`.

Initialization

Trying to make rules more general and to minimize spurious typing has a long tradition in C++ and most other languages. I suspect it is a never-ending task, but of course the desire for uniformity of rules and simple notation is not restricted to trivial examples. In particular, every irregularity is a source of complexity, added learning time, and bugs (when you make the wrong choice among alternatives). Irregularity also becomes a barrier to generic programming, which critically depends on identical notation for a set of types. One area where uniformity and generality is seriously lacking in C++98 is initialization. There is a bewildering variety of syntaxes (`{...}`, `(...)`, `=...`, or `default`), semantics (e.g., copy or construct), and applicability (can I use a `{...}` initializer for `new`? for a `vector`? Can I use a `(...)` initializer for a local variable?). Some of this irregularity goes all the way back to the early days of C, but C++0x manages to unify and generalize all of these mechanisms: You can use `{...}` initialization everywhere. For example:

```
int v1 = 7;
```

```

int v2(7);
int v3 = { 7 }; // yes that's legal C and C++98
int v4 {7}; // new for C++0x: initialize v4 with 7

int x1; // default x1 becomes 0 (unless x1 is a local variable)
int x2(); // oops a function declaration
int x3 = {}; // new for C++0x: give x3 the default int value (0)
int x4{}; // new for C++0x: give x4 the default int value (0)

```

What could be simpler than initializing an **int**? Yet, we see surprises and irregularities. This gets much more interesting when we consider aggregates and containers:

```

int a[] = { 1, 2, 3 };
S s = { 1, 2, 3 }; // ok, maybe
std::vector<int> v1 = { 1, 2, 3 }; // new in C++0x
std::vector<int> v2 { 1, 2, 3 }; // new in C++0x

```

It always bothered me that we couldn't handle **v1** in C++. That violated the principle that user-defined and built-in types should receive equivalent support (so why does my favorite container, **vector**, get worse support than the perennial problem, array?). It also violated the principle that fundamental operations should receive direct support (what's more fundamental than initialization?).

When does **S s = { 1, 2, 3 };** actually work? It works in C++98 iff **S** is a **struct** (or an array) with at least three members that can be initialized by an **int** and iff **S** does not declare a constructor. That answer is too long and depends on too many details about **S**. In C++0x, the answer boils down to "iff **S** can be initialized by three integers." In particular, that answer does not depend on how that initialization is done (e.g. constructors or not). Also, the answer does not depend on exactly what kind of variable is being initialized. Consider:

```

int a[] = { 1, 2, 3 };
void f1(const int(&)[3]); // reference to array
f1({1,2,3}); // new in C++0x
p1 = new int[]{1,2,3}; // new in C++0x

S s = { 1, 2, 3 }; // ok, maybe
void f2(S);
f2({1,2,3}); // new in C++0x
p2 = new S {1,2,3}; // new in C++0x

std::vector<int> v2 { 1, 2, 3 }; // new in C++0x
void f3(std::vector<int>);
f3({ 1, 2, 3 }); // new in C++0x
p3 = new std::vector{1,2,3}; // new in C++0x

```

Why didn't I just say `void f1(int[])`? Well, compatibility is hard to deal with. That `[]` "decays" to `*` so unfortunately `void f1(int[])` is just another way of saying `void f1(int*)` and we can't initialize a pointer with a list. Array decay is the root of much evil.

I don't have the time or space to go into details, but I hope you get the idea that something pretty dramatic is going on here: We are not just adding yet another mechanism for initialization, C++0x provides a single uniform mechanism for initializing objects of all types. For every type `X`, and wherever an initialization with a value `v` makes sense, we can say `X{v}` and the resulting object of type `X` will have the same value in all cases. For example:

```
X x{v};
X* p = new X{v};
auto x2 = X{v};           // explicit conversion
void f(X);
f(X{v});
f({v});
struct C : X {
    C() : X{v}, a{v} { }
    X a;
    // ...
};
```

Have fun imagining how to do that using C++98 for all types you can think of (note that `X` might be a container, `v` might be a list of values or empty). One of my favorites is where `X` is a `char*` and `v` is `7`.

Support for low-level programming

There is a school of thought that all programming these days is "web services" and that "efficiency" ceased to matter decades ago. If that's your world, the lower reaches of C++ are unlikely to be of much interest. However, most computers these days are embedded, processors are not getting any faster (in fact many are getting slower as multi-cores become the norm), and all those web-services and high-level applications have to be supported by efficient and compact infrastructure components. So, C++0x has facilities to make C++ a better language for low-level systems programming (and for programming aiming for efficiency in general).

At the bottom of everything in a computer is memory. Working directly with physical memory is unpleasant and error prone. In fact, if it wasn't for the C++0x memory model (see below) it would be impossible for humans to write code at the traditional "C language level." Here I'm concerned about how to get from the world of `ints`, arrays, and pointers to the first level of abstraction.

Plain Old data and layout

Consider

```
struct S1 {
    int a, b;
```

```
};

struct S2 {
    S2(int aa, int bb) : a{aa}, b{bb} { }
    S2() { } // leave a and b uninitialized
    int a,b;
};
```

S1 and **S2** are “standard-layout classes” with nice guarantees for layout, objects can be copied by `memcpy()` and shared with C code. They can even be initialized by the same `{...}` initializer syntax. In C++98, that wasn’t so. They required different initialization syntaxes and only **S1** had the nice guarantees, but **S2** simply wasn’t a POD (the C++98 term for “guaranteed well-behaved layout”).

Obviously, this improvement shouldn’t be oversold as “major,” but constructors are really nice to have and you could already define ordinary member functions for a POD. This can make a difference when you are close to the hardware and/or need to interoperate with C or Fortran code. For example, `complex<double>` is a standard-layout class in C++0x, but it wasn’t in C++98.

Unions

Like `structs`, C++98 `unions` had restrictions that made them hard to use together with the abstraction mechanisms, making low-level programming unnecessarily tedious. In particular, if a class had a constructor, a destructor, or a user-defined copy operation, it could not be a member of a `union`. Now, I’m no great fan of `unions` because of their misuses in higher-level software, but since we have them in the language and need them for some “close to the Iron” tasks, the restrictions on them should reflect reality. In particular, there is no problem with a member having a destructor; the real problem is that if it has one it must be invoked iff it is the member used when the union is destroyed (iff the union is ever destroyed). So the C++0x rules for a union are:

- No virtual functions (as ever)
- No references (as ever)
- No bases (as ever)
- If a union has a member with a user-defined constructor, copy, or destructor then that special function is “deleted;” that is, it cannot be used for an object of the union type.

This implies that we can have a `complex<double>` as a union member. It also implies that if we want to have members with user-defined copy operations and destructors, we have to implement some kind of variant type (or be very careful about how we use the objects of the unions or the compiler will catch us). For example:

```
class Widget {           // Three alternative implementations represented as a union
private:
    enum class Tag { point, number, text } type;    // discriminant

    union { // compact representation
        point p; // point has constructor
        int i;
```

```

        string s; // string has default constructor, copy operations, and destructor
    };

    // ...

    Widget& operator=(const Widget& w) // necessary because of the string union member
    {
        if (type==Tag::text && w.type==Tag::text) {
            s = w.s; // usual string assignment
            return *this;
        }

        if (type==Tag::text) s.~string(); // destroy (explicitly!)

        switch (type=w.type) {
            case Tag::point: p = w.p; break; // normal copy
            case Tag::number: i = w.i; break;
            case Tag::text: new(&s)(w.s); break; // placement new
        }
        return *this;
    }
};

```

I'm still not a great fan of unions. It often takes too much cleverness to use them right. However, I don't doubt that they have their uses and C++0x serves their users better than C++98.

Did you notice the “**enum class**” notation above? An **enum class** is an **enum** where the enumerators have class scope and where there is no implicit conversion to **int**. I used a **class enum** (a “strongly typed **enum**”) here, so that I could use tag names without worrying about name. Also, not the use of **Tag::**, in C++0x we can qualify an enumerator with the name of its enumeration for clarity or disambiguation.

General constant expressions

Compile-time evaluation can be a major saver of run time and space. C++98 offers support for some pretty fancy template meta-programming. However, the basic constant expression evaluation facilities are somewhat impoverished: We can do only integer arithmetic, cannot use user-defined types, and can't even call a function. C++0x takes care of that by a mechanism called **constexpr**:

- We can use floating-point in constant expressions
- We can call simple functions (“**constexpr** functions”) in constant expressions
- We can use simple user-defined types (“literal types”) in constant expressions
- We can request that an expression must be evaluated at compile time

For example:

```

enum Flags { good=0, fail=1, bad=2, eof=4 };

constexpr int operator|(Flags f1, Flags f2) { return Flags(f1|f2); }

void f(Flags x)

```

```

{
    switch (x) {
        case bad:      /* ... */ break;
        case eof:      /* ... */ break;
        case bad|eof:  /* ... */ break;
        default:       /* ... */ break;
    }
}

```

Here **constexpr** says that the function must be of a simple form so that it can be evaluated at compile time when given constant expressions as arguments. That “simple form” is a single return statement, so we don’t do loops or declare variables in a **constexpr** function, but since we do have recursion the sky is the limit. For example, I have seen a very useful integer square root **constexpr** function.

In addition to be able to evaluate expressions at compile time, we want to be able to *require* expressions to be evaluated at compile time; **constexpr** in front of a variable definition does that (and implies **const**):

```

constexpr int x1 = bad|eof; // ok

void f(Flags f3)
{
    constexpr int x2 = bad|f3; // error: can't evaluate at compile time
    const int x3 = bad|f3;     // ok: but evaluated at run time
    // ...
}

```

Typically we want the compile-time evaluation guarantee for one of two reasons:

- For values we want to use in constant expressions (e.g. case labels, template arguments, or array bounds)
- For variables in namespace scope that don’t want run-time initialized (e.g. because we want to place them in read-only storage).

This also works for objects for which the constructors are simple enough to be **constexpr** and expressions involving such objects:

```

struct Point {
    int x,y;
    constexpr Point(int xx, int yy) : x{xx}, y{yy} { }
};

constexpr Point origo { 0,0 };
constexpr int z { origo.x };

constexpr Point a[] { Point{0,0}, Point{1,1}, Point{2,2} };
constexpr int x { a[1].x }; // x becomes 1

```

Who needs this? Why isn’t “good old **const**” good enough? Or conversely, as one academic in all seriousness asked, “Why don’t you just provide a full compile-time interpreter?” I think the answers to the first two questions are interesting and relate to general principles. Think: what do people do when

they have only “good old **const**”? They hit the limits of **const** and proceed to use macros for examples like the **Flags** one. In other words, they fall back on typeless programming and the most error-prone abstraction mechanism we know. The result is bugs. Similarly, in the absence of compile-time user-defined types, people revert to a pre-classes style of programming which obscures the logic of what they are doing. Again, the result is bugs.

I observed two other phenomena:

- Some people were willing to go to extremes of cleverness to simulate compile-time expression evaluation: Template instantiation is Turing complete, but you do have to write rather “interesting” code to take advantage of that in many cases. Like most workarounds, it’s much more work (for programmers and compilers) than a specific language feature and with the complexity comes bugs and learning time.
- Some people (mostly in the embedded systems industry) have little patience with Turing completeness or clever programming techniques. On the other hand, they have a serious need to use ROM, so special-purpose, proprietary facilities start to appear.

The **constexpr** design is interesting in that it addresses some serious performance and correctness needs by doing nothing but selectively easing restrictions. There is not a single new semantic rule here: it simply allows more of C++ to be used at compile time.

Narrowing

Maybe you noticed that I used {} initialization consistently. Maybe, you also thought that I was uglifying code and was overenamored by a novel feature? That happens, of course, but I don’t think that is the case here. Consider:

```
int x1 = 64000;
int x2 { 64000 };
```

We can have a nice friendly discussion about the aesthetics of those two definitions and you might even point out that the {} version requires one more keystroke than the = one. However, there is one significant difference between those two forms that makes me chose {}. The {} version doesn’t allow narrowing and I failed to tell you that the two definitions were written for a machine with 16-bit **ints**. That means that the value of **x1** could be very surprising, whereas the definition of **x2** causes a compile-time error.

When you use {} initializers, no narrowing conversions are allowed:

- No narrowing integral conversions (e.g., no **int-to-char** conversion)
- No floating-to-integral conversions (e.g. no **double-to-int** or **double-to-bool** conversion)
- When an initializer is a constant expression the actual value is checked, rather than the type (e.g. **char c {'x'};** is ok because 'x' fits in a **char**)

This directly addresses a source of nasty errors that has persisted since the dawn of C.

Tools for writing classes

Most of what is good, efficient, and effective about C++ involves designing, implementing, and using classes. Thus, anything that is good for classes is good for C++ programmers. People often look for “major features” and “solutions,” but I think of language features as building blocks: If you have the right set of building blocks, you can apply them in combination to provide “solutions.” In particular, we need simple and powerful “building blocks” to provide general or application-specific abstractions in the form of classes and libraries. Thus, “little features” that are unimportant in themselves may have a major impact as one of a set of interrelated features. When designed well, the sum is indeed greater than the

parts. Here, I will just give three examples: initializer-list constructors, inheriting constructors, and move semantics.

Initializer list constructors

How did we manage to get a `std::vector` to accept a list of elements as its initializer? For example:

```
vector<int> v0 {};           // no elements
vector<int> v1 { 1 };      // one element
vector<int> v2 { 1,2 };    // two elements
vector<int> v3 { 1,2,3 };  // three elements
vector<int> v4 { 1, 2, 3, 4, 5, a, b, c, d, x+y, y*z, f(1,d) }; // many elements
```

That's done by providing `vector` with an “initializer-list constructor”:

```
template<class T> class vector {
    vector(std::initializer_list<T> a)    // initializer-list constructor
    {
        reserve(a.size());
        uninitialized_copy(a.begin(),a.end(),v.begin());
    }
    // ...
};
```

Whenever the compiler sees a `{ ... }` initializer for a vector, it invokes the initializer constructor (only if type checking succeeds, of course). The `initializer_list` class is known to the compiler and whenever we write a `{ ... }` list that list will (if possible) be represented as an `initializer_list` and passed to the user's code. For example:

```
void f(int,initializer_list<int>,int);
f(1,{1,2,3,4,5},1);
```

Of course it is nice to have a simple and type safe replacement for the `stdargs` macros, but the serious point here is that C++ is moving closer to its stated ideal of uniform support of built-in types and user-defined types [Stroustrup,1994]. In C++98, we could not build a container (e.g., `vector`) that was as convenient to use as a built-in array; now we can.

Inheriting constructors

In C++98, we can derive a class from another, inheriting the members. Unfortunately, we can't inherit the constructors because if a derived class adds members needing construction or virtual function (requiring a different virtual function table) then its base constructors are simply wrong. This restriction – like most restrictions – can be a real bother; that is, the restriction is a barrier to elegant and effective use of the language. My favorite example is a range checked vector:

```
template<class T> class Vec : public std::vector<T> {
public:
```

```

    using vector<T>::vector;    // use the constructors from the base class
    T& operator[](size_type i) { return this->at(i); }
    const T& operator[](size_type i) const { return this->at(i); }
};

```

The solution, to provide an explicit way of “lifting up” constructors from the base, is identical to the way we (even in C++98) bring functions from a base class into the overload set of a derived class:

```

struct B {
    void f(int);
    void f(double);
};

struct D : B {
    using B::f;                // “import” f()s from B
    void f(complex<double>);   // add an f() to the overload set
};

D x;
x.f(1);                       // B::f(int);
x.f({1.3,3.14});              // D::f(complex<double>);

```

Thus, this language extension is really a generalization and will in most contexts simplify the learning of use of C++. Note also the use of the {...} initialization mechanism in that last example: Of the alternatives, only `f(complex<double>)` can be initialized with a pair of doubles, so the {...} notation is unambiguous.

Move semantics

Consider

```

template<class T> void swap(T& a, T& b)
{
    T tmp = a;    // copy a into tmp
    a = b;       // copy b into a
    b = tmp;     // copy tmp into b
}

```

But I didn’t want to copy anything, I wanted to swap! Why do I have to make three copies? What if `T` is something big and expensive to copy? This is a simple and not unrealistic example of the problem with copy: After each copy operation, we have two copies of the copied value and often that’s not really what we wanted because we never again use the original. We need a way of saying that we just want to move a value!

In C++0x, we can define “move constructors” and “move assignments” to move rather than copy their argument:

```

template<class T> class vector {

```

```

// ...
vector(const vector&);           // copy constructor
vector(vector&&);               // move constructor
vector& operator=(const vector&); // copy assignment
vector& operator=(vector&&);    // move assignment
};

```

The “&&” means “rvalue reference.” An rvalue reference is a reference that can bind to an rvalue. The point about an rvalue here is that we may assume that it will never be used again after we are finished with it. The obvious implementation is for the **vector** move constructor it therefore to grab the representation of its source and to leave its source as the empty **vector**. That is often far more efficient than making a copy. When there is a choice, that is, if we try to initialize or assign from an rvalue, the move constructor (or move assignment) is preferred over the copy constructor (or copy assignment). For example:

```

vector<int> make_rand(int s)
{
    vector<int> res(s);
    for (auto& x : res) x = rand_int();
    return res;
}
vector<int> v { make_rand(10000) };

void print_rand(int s)
{
    for (auto x : make_rand(s)) cout << x << '\n';
}

```

Obviously (once you become used to thinking about moves), no **vector** is copied in this example. If you have ever wanted to efficiently return a large object from a function without messing with free store management, you see the point.

So what about **swap()**? The C++0x standard library provides:

```

template<class T>
void swap(T& a, T& b)    // “perfect swap” (almost)
{
    T tmp = std::move(a);
    a = std::move(b);
    b = std::move(tmp);
}

```

The standard library function **move(x)** means “you may treat **x** as an rvalue.” Rvalue references can also be used to provide perfect forwarding: A template function **std::forward()** supports that. Consider a

“factory function” that given an argument has to create an object of a type and return a **unique_ptr** to the created object:

```
template <class T, class A1> std::unique_ptr<T> factory(A1&& a1)
{
    return std::unique_ptr<T>{new T{std::forward<A1>{a1}}};
}

unique_ptr p1 { factory<vector<string>>{100} } }
```

A **unique_ptr** is a standard-library “smart pointer” that can be used to represent exclusive ownership; it is a superior alternative to **std::shared_ptr** in many (most?) cases.

One way to look at rvalue references is that C++ had a pressing need to support move semantics: Getting large values out of functions was at best inelegant or inefficient, transferring ownership by using shared pointers was logically wrong, and some forms of generic programming cause people to write a lot of forwarding functions that in principle implies zero run-time costs but in reality were expensive. Sometimes move vs. copy is a simple optimization and sometimes it is a key design decision, so a design must support both. The result was rvalue references, which supports the name binding rules that allows us to define **std::move()**, **std::forward()**, and the rules for copy and move constructors and assignments.

User-defined literals

For each built-in type we have corresponding literals:

```
'a'      '\n'      // char
1        345      // int
345u     // unsigned
1.2f     // float
1.2      12.345e-7 // double
"Hello, world!" // C-style string
```

However, C++98 does not offer an equivalent mechanism for user-defined types:

```
1+2i     // complex
"Really!"s // std::string
103F 39.5c // Temperature
123.4567891234df // decimal floating point
123s     // seconds
101010111000101b // binary
1234567890123456789012345678901234567890x // extended-precision
```

Not providing literals for user-defined types is clear violation of the principle that user-defined and built-in types should receive equivalent support. I don't see how C++ could have survived without user-defined literals! Well, I do: inlined constructors is a pretty good substitute and **constexpr** constructors would be even better, but why not simply give people the notation they ask for?

C++0x supports “user-defined literals” through the notion of literal operators that map literals with a given suffix into a desired type. For example:

```
constexpr complex<double> operator"" i(long double d) // imaginary literal
{
    return {0,d}; // complex is a literal type
}

std::string operator "" s(const char* p, size_t n) // std::string literal
{
    return string{p,n}; // requires free store allocation
}
```

Note the use of **constexpr** to enable compile-time evaluation; **complex<double>** is a literal type. The syntax is **operator""** to say that a literal operator is being defined followed by the name of the function, which is the suffix to be recognized. Given those literal operators, we can write:

```
template<class T> void f(const T&);
f("Hello"); // pass pointer to char*
f("Hello"s); // pass (5-character) std::string object
f("Hello\n"s); // pass (6-character) std::string object

auto z = 2+3.14i; // 2+complex<double>(0,3.14)
```

The basic (implementation) idea is that after parsing what could be a literal, the compiler always checks for a suffix. The user-defined literal mechanism simply allows the user to specify a new suffix and what is to be done with the literal before it. It is not possible to redefine the meaning of a built-in literal suffix or invent new syntax for literals. In this, user-defined literals are identical to user-defined operators.

A literal operator can request to get its (preceding) literal passed “cooked” (with the value it would have had if the new suffix hadn’t been defined) or “raw” (the string of characters exactly as typed).

To get an “uncooked” string, simply request a **const char*** argument:

```
Bignum operator"" x(const char* p)
{
    return Bignum(p);
}

void f(Bignum);
f(1234567890123456789012345678901234567890x);
```

Here the C-style string "12345678901234567890123456789012345678901234567890" is passed to **operator"" x()**. Note that we did not have to explicitly put those digits into a string, though we could have:

```
f("1234567890123456789012345678901234567890"x);
```

Note that “literal” does not mean “efficient” or “compile-time evaluated.” If you need run-time performance, you can design for that, but “user-defined literals” is – in the best C++ tradition – a very general mechanism.

The sum of language extensions

When you add it all up, C++0x offers are many new language facilities. The C++0x FAQ list them:

- [__cplusplus](#)
- [alignments](#)
- [attributes](#)
- [atomic operations](#)
- [auto](#) (type deduction from initializer)
- [C99 features](#)
- [enum class](#) (scoped and strongly typed **enums**)
- [copying and rethrowing exceptions](#)
- [constant expressions](#) (generalized and guaranteed; **constexpr**)
- [decltype](#)
- default template parameters for function
- [defaulted and deleted functions](#) (control of defaults)
- [delegating constructors](#)
- [Dynamic Initialization and Destruction with Concurrency](#)
- [explicit conversion operators](#)
- extended **friend** syntax
- [extended integer types](#)
- [extern templates](#)
- [for statement](#); see range **for** statement
- generalized SFINAE rules
- [in-class member initializers](#)
- [inherited constructors](#)
- [initializer lists](#) (uniform and general initialization)
- [lambdas](#)
- [local classes as template arguments](#)
- [long long integers](#) (at least 64 bits)
- [memory model](#)
- [move semantics](#); see [rvalue references](#)
- [Namespace Associations](#) (Strong using)
- [Preventing narrowing](#)
- [null pointer](#) (**nullptr**)
- [PODs](#) (generalized)
- [range for statement](#)
- [raw string literals](#)
- [right-angle brackets](#)
- [rvalue references](#)
- [static \(compile-time\) assertions](#) (**static_assert**)
- [suffix return type syntax](#) (extended function declaration syntax)
- [template alias](#)

- [template typedef](#); see template alias
- [thread-local storage](#) (**thread_local**)
- [unicode characters](#)
- [Uniform initialization syntax and semantics](#)
- [unions](#) (generalized)
- [user-defined literals](#)
- [variadic templates](#)

Fortunately most are minor and much work has been spent to ensure that they work in combination. The sum is greater than its parts.

Like inheriting constructors, some of the new features address fairly specific and localized problems (e.g. raw literals for simpler expression of regular expressions, **nullptr** for people who are upset by **0** as the notation for the null pointer, and **enum classes** for stronger type-checked enumerations with scoped enumerators). Other features, like general and uniform initialization, aim to support more general programming techniques (e.g. **decltype**, variadic templates, and template aliases for the support of generic programming). The most ambitious new support for generic programming, concepts, didn't make it into C++0x (see [Stroustrup,2009]).

When exploring those new features (my C++0x FAQ is a good starting point), I encourage you to focus on how they work in conjunction with old and new language features and libraries. I think of these language features as “building bricks” (my home town is about an hour's drive from the Lego factory) and few make much sense when considered in isolation.

Concurrency and memory model

Concurrency has been the next big thing for about 50 years, but concurrency is no longer just for people with multi-million dollar equipment budgets. For example, my cell phone (programmed in C++ of course) is a multi-core. We don't just have to be able to do concurrent programming in C++ (as we have “forever”), we need a standard for doing so and help to get concurrent code general and portable. Unfortunately, there is not just one model for concurrency and just one way of writing concurrent code, so standardization implies serious design choices.

Concurrency occurs at several levels in a system, the lowest level visible to software is the level of individual memory accesses. With multiple processors (“cores”) sharing a memory hierarchy of caches, this can get quite “interesting.” This is the level addressed by the memory model. The next level up is the systems level where computations are represented by threads. Above that can be general or application-specific models of concurrency and parallel computation.

The general approach of C++0x is to specify the memory model, to provide primitive operations for dealing with concurrency, and to provide language guarantees so that concurrency mechanism, such as threads, can be provided as libraries. The aim is to enable support for a variety of models of concurrency, rather than building one particular one into the language.

The memory model

The memory model is a treaty between the machine architects and the compiler writers to ensure that most programmers do not have to think about the details of modern computer hardware. Without a memory model, few things related to threading, locking, and lock-free programming would make sense.

The key guarantee is: Two threads of execution can update and access separate memory locations without interfering with each other. To see why that guarantee is non-trivial, consider:

```
// thread 1:  
char c;  
c = 1;  
int x = c;  
  
// thread 2:  
char b;  
b = 1;  
int y = b;
```

For greater realism, I could have used separate compilation (within each thread) to ensure that the compiler/optimizer won't simply ignore **c** and **b** and directly initialize **x** and **y** with 1. What are the possible values of **x** and **y**? According to C++0x, the only correct answer is the obvious one: 1 and 1. The reason that's interesting is that if you take a conventional good pre-concurrency C or C++ compiler, the possible answers are 0 and 0, 1 and 0, 0 and 1, and 1 and 1. This has been observed "in the wild." How? A linker might allocate **c** and **b** in the same word – nothing in the C or C++ 1990s standards says otherwise. In that, C and C++ resemble all languages not designed with real concurrent hardware in mind. However, most modern processors cannot read or write a single character, a processor must read or write a whole word, so the assignment to **c** really is "read the word containing **c**, replace the **c** part, and write the word back again." Since the assignment to **b** is similarly implemented, there are plenty of opportunities for the two threads to clobber each other even though the threads do not (according to their source text) share data!

So naturally, C++0x guarantees that such problems do not occur for "separate memory locations." In this example, **b** and **c** will (if necessary on a given machine) be allocated in different words. Note that different bitfields within a single word are not considered separate memory locations, so don't share structs with bitfields among threads without some form of locking. Apart from that caveat, the C++ memory model is simply "as everyone would expect."

Fortunately, we have already adapted to modern times and every current C++ compiler (that I know of) gives the one right answer and has done so for years. After all, C++ has been used for serious systems programming of concurrent systems "forever."

Threads, locks, and atomics

In my opinion, letting a bunch of threads loose in a shared address space and adding a few locks to try to ensure that the threads don't stomp on each other is just about the worst possible way of managing concurrency. Unfortunately, it is also by far the most common model and deeply embedded in modern systems. To remain a systems programming language, C++ must support that style of programming, and support it well, so C++0x does. If you know Posix threads or boost threads, you have a first-order

approximation of what C++0x offers at the most basic level. To simplify the use of this fundamental (and flawed) model of concurrency, C++0x also offers

- thread local storage (identified by the keyword **thread_local**)
- mutexes
- locks
- conditions variables
- a set of atomic types for lock-free programming and the implementation of other concurrency facilities
- a notion of fine grain time duration

In my opinion lock-free programming is a necessity, but should be reserved for people who find juggling naked sharp swords too tame [Dechev,2009]. Importantly, the whole language and standard library has been re-specified (down to the last memory read or write) so that the effects of concurrency are well-specified – though of course not well defined: the result of a data race is not and should not be well defined (it should be prevented by the library or applications programmer).

As luck would have it, Anthony Williams has a paper “Multi-threading in C++0x” in the current issue of Overload [Williams, 2009], so I don’t have to go into details. Instead, I will give an example of a way for the programmer to rise above the messy threads-plus-lock level of concurrent programming:

```

template<class T, class V> struct Accum { // function object type for computing sums
    T* b;
    T* e;
    V val;
    Accum(T* bb, T* ee, const V& v) : b{bb}, e{ee}, val{vv} {}
    V operator() () { return std::accumulate(b,e,val); }
};

void comp(vector<double>& v)
    // spawn many tasks if v is large enough
{
    if (v.size()<10000) return std::accumulate(v.begin(),v.end(),0.0);

    auto f0 {async(Accum{v.data(), v.data()+v.size()/4, 0.0})};
    auto f1 {async(Accum{v.data()+v.size()/4, v.data()+v.size()/2, 0.0})};
    auto f2 {async(Accum{v.data()+v.size()/2, v.data()+v.size()*3/4, 0.0})};
    auto f3 {async(Accum{v.data()+v.size()*3/4, v.data()+v.size(), 0.0})};

    return f0.get()+f1.get()+f2.get()+f3.get();
}

```

This is a very simple-minded use of concurrency (note the “magic number”), but note the absence of explicit threads, locks, buffers, etc. The type of the **f**-variables are determined by the return type of the standard-library function **async()** which called a **future**. If necessary, **get()** on a **future** waits for a

std::thread to finish. Here, it is **async()**'s job to spawn **threads** as needed and the **future**'s job to **join()** the appropriate **threads** (i.e., wait for the completion of threads) . “Simple” is the most important aspect of the **async()/future** design; **futures** can also be used with threads in general, but don't even *think* of using **async()** to launch tasks that do I/O, manipulate mutexes, or in other ways interact with other tasks. The idea behind **async()** is the same as the idea behind the **range-for** statement: Provide a simple way to handle the simplest, rather common, case and leave the more complex examples to the fully general mechanism.

Please note that **future** and **async()** is just one example of how to write concurrent programs above the messy threads-plus-lock level. I hope to see many libraries supporting a variety of models, some of which might become candidates for C++1x. Unlike every other feature presented here, **async()** has not yet been voted into C++0x . That's expected to happen in October, but no man's life, liberty, or programming language is safe while the committee is in session (apologies to Mark Twain). [It was voted in.]

Standard Library Improvements

At the outset of the work on C++0x, I stated my ideal as “being careful, deliberate, conservative and skeptic” about language extensions, but “opportunistic and ambitious” about new standard libraries [Stroustrup,2002] . At first glance, the opposite happened, but when you count pages in the standard you find that the language sections grew by about 27% and the library sections by about 100%, so maybe it would be wrong to complain too loudly about lack of new standard library components. The most obvious improvements to the standard library are the added library components:

- Concurrency ABI:
 - **thread**
 - mutexes, locks, atomic types,
 - simple asynchronous value exchange: **future**, **shared_future**, **atomic_future**, and **promise**
 - simple asynchronous launcher: **async()**
- Containers:
 - Hashed containers: **unordered_map**, **unordered_multimap**, **unordered_set**, **unordered_multiset**
 - Fixed sized array: **array**
 - Singly-linked list: **forward_list**
- Regular expressions: **regex**
- Random numbers
- Time utilities: **duration** and **time_point**
- Compile-time rational arithmetic: **ratio**
- Resource management pointers: **unique_ptr**, **shared_ptr**, and **weak_ptr**
- Utility components: **bind()**, **function**, **tuple**
- Metaprogramming and type traits
- Garbage collection ABI

Whatever project you do, one or more of these libraries should help. As usual for C++, the standard libraries tend to be utility components rather than complete solutions to end-user problems. That makes them more widely useful.

Another set of library improvements are “below the surface” in that they are improvements to existing library components rather than new components. For example, the C++0x **vector** is more flexible and more efficient than the C++98 **vector**. As usual, the standard library is the first test of new language features: If a language feature can’t be used to improve the standard library, what is it good for?

More containers

So, C++0x gets hash tables (**unordered_map**, etc.), a singly-linked list (**forward_list**), and a fixed-sized container (**array**). What’s the big deal and why the funny names? The “big deal” is simply that we have standard versions, available in every C++0x implementation (and in major C++ implementations today), rather than having to build, borrow, or buy our own. That’s what standardization is supposed to do for us. Also, since “everybody” has written their own version (proving that the new components are widely useful), the standard could not use the “obvious” names: there were simply too many incompatible **hash_maps** and **slists** “out there,” so new names had to be found: “unordered” indicates that you can’t iterate over an **unordered_map** in a predictable order defined by <; an **unordered_map** uses a hash function rather than a comparison to organize elements. Similarly, it is a defining characteristic of a **forward_list** (a singly linked list) that you can iterate through it forwards (using a forward iterator), but not (in any realistic way) backwards.

The most important point about **forward_list** is that it is more compact (and has slightly more efficient operations) than **list** (a doubly-linked list): An empty **forward_list** is one word and a link has only a one-word overhead. There is no **size()** operation, so the implementation doesn’t have to keep track of the size in an extra word or (absurdly) count the number of elements each time you innocently ask.

The point of **unordered_map** and its cousins is runtime performance. With a good hash function, lookup is amortized $O(1)$ as compared to **map**’s $O(\log(N))$, which isn’t bad for smaller containers. Yes, the committee cares about performance.

Built-in arrays have two major problems: They implicitly “decay” to pointers at the slightest provocation and once that has happened their size is “lost” and must be “managed” by the programmer. A huge fraction of C and C++ bugs have this as their root cause. The standard-library **array** is most of what the built-in array is without those two problems. Consider:

```
array<int,6> a { 1, 2, 3 };
a[3] { 4 };
int x { a[5] };           // x becomes 0 because default elements are zero initialized
int* p1 { a };           // error: std::array doesn't implicitly convert to a pointer
int* p2 { a.data() };    // ok: get pointer to first element
```

Unfortunately you cannot deduce the length of an **array** from an initializer list:

```
array<int> a3 { 1, 2, 3 }; // error: size unknown/missing
```

That's about the only real advantage left for built-in arrays over **std::array**.

The standard **array**'s features make it attractive for embedded systems programming (and similar constrained, performance-critical, or safety-critical tasks). It is a sequence container so it provides the usual member types and functions (just like **vector**). In particular, **std::array** knows its **end()** and **size()** so that “buffer overflow” and other out-of-range access problems are easily avoided. Consider:

```
template<class C, class V> typename C::const_iterator find(const C& a, V val)
{
    return find(a.begin(), a.end(), val);
}

array<int,10> a10;
array<double,1000> a1000;
vector<int> v;
// ...
auto answer = find(a10,42);
auto cold = find(a1000,-274.15);
if (find(v,666)==v.end()) cout << "See no evil";
```

Incidentally, have you ever been annoyed by having to write things like **typename C::const_iterator**? In C++0x, the compiler can deduce the return type of a simple function from its **return**-statement, so you can simplify:

```
template<class C, class V> [] find(const C& a, V val)
{
    return find(a.begin(), a.end(), val);
}
```

You can read [] as “function;” [] is a new notation to explicitly state that a function is being declared. [Mark Twain strikes again: This has now become uncertain if that last example can be written that way or should be slightly different. It is the last such question.]

Better containers

I suspect that the new containers will attract the most attention, but the “minor improvements” to the existing containers (and other standard library components) are likely to be the more important.

Initializer lists

The most visible improvement is the use of initializer-list constructors to allow a container to take an initializer list as its argument:

```
Vector<string> vs = { "Hello", "", "", "World!", "\n" };
for (auto s : vs ) cout << s;
```

This is shorter, clearer, and potentially more efficient than building the vector up element by element. It gets particularly interesting when used for nested structures:

```
vector<pair<string,Phone_number>> phone_book= {
    { "Donald Duck", 2015551234 },
    { "Mike Doonesbury", 9794566089 },
    { "Kell Dewclaw", 1123581321 }
};
```

As an added benefit, we get the protection from narrowing from the { }-notation, so that if any of those integer values do not fit into **Phone_number**'s representation of them (say, a 32-bit **int**), the example won't compile.

Move operators

Containers now have move constructors and move assignments (in addition to the traditional copy operations). The most important implication of this is that we can efficiently return a container from a function:

```
vector<int> make_random(int n)
{
    vector<int> ref(n);
    for(auto x& : ref) x = rand_int();    // some random number generator
    return ref;
}

vector<int> v = make_random(10000);
for (auto x : make_random(1000000)) cout << x << '\n';
```

The point here is that – despite appearances – no **vector** is copied. In C++98, this **make_random()** is a performance problem waiting to happen; in C++0x it is an elegant direct solution to a classic problem. Consider the usual workarounds: Try to rewrite **make_random()** to return a free-store-allocated **vector** and you have to deal with memory management. Rewrite **make_random()** to pass the **vector** to be filled as an argument and you have far less obvious code (plus an added opportunity for making an error).

Improved push operations

My favorite container operation is **push_back()** that allows a container to grow gracefully:

```
vector<pair<string,int>> vp;
string s;
int i;
while(cin>>s>>i) vp.push_back({s,i});
```

This will construct a **pair<string,int>** out of **s** and **i** and move it into **vp**. Note: “move” not “copy.” There is a **push_back** version that takes an rvalue reference argument so that we can take advantage of **string**'s move constructor. Note also the use of the {}-list syntax to avoid verbosity.

Emplace operations

The `push_back()` using a move constructor is far more efficient than the traditional copy-based one in important cases, but in extreme cases we can go further. Why copy/move anything? Why not make space in the vector and then construct the desired value in that space? Operations that do that are called “emplace” (meaning “putting in place”). For example `emplace_back()`:

```
vector<pair<string,int>> vp;
string s;
int i;
while(cin>>s>>i) vp.emplace_back(s,i);
```

An emplace function takes a variadic template (see the C++0xFAQ) argument and uses that to construct an object of the desired type in place. Whether the `emplace_back()` really is more efficient than the `push_back()` depends on types involved and the implementation (of the library and of variadic templates). In this case, there doesn't seem to be a performance difference. As ever, if you think it might matter, measure. Otherwise, choose based on aesthetics: `vp.push_back({s,i})` or `vp.emplace_back(s,i)`. For now, I prefer the `push_back()` version because I can see that an object is being composed, but that might change over time. For new facilities, it is not immediately obvious which styles and which combinations will be the more effective and more maintainable.

Scoped allocators

Containers can now hold “real allocation objects” (with state) and use those to control nested/scoped allocation (e.g. allocation of elements in a container). A rather sneaky problem can occur when using containers and user-defined allocators: Should an element's free-store allocated sub-objects be in the same allocation area as its container? For example, if you use `Your_allocator` for `Your_string` to allocate its elements and I use `My_allocator` to allocate elements of `My_vector` then which allocator should be used for string elements in `My_vector<Your_string>`? The solution is to tell a container when to pass an allocator to an element. For example, assuming that I have an allocator `My_alloc` and I want a `vector` that uses `My_alloc` for both the `vector` element and `string` element allocations. First, I must make a version of `string` that accepts `My_alloc` objects:

```
using xstring = basic_string< // a string with my allocator
                           char,
                           char_traits<char>,
                           My_alloc<char>
                           >;
```

This use of `using` is new in C++0x. It is basically a variant of `typedef` that allows us to define an alias with the name being defined coming up front where we can see it.

Next, I must make a version of `vector` that accepts those `xstrings`, accepts a `My_alloc` object, and passes that object on to the `xstring`:

```
using svec = vector< // a string with a scoped allocator
                  xstring,
                  scoped_allocator_adaptor<My_alloc<xstring>>
                  >;
```

The standard library “adaptor” (“wrapper type”) **scoped_allocator_adaptor** is used to indicate that **xstring** also should use **My_alloc**. Note that the adaptor can (trivially) convert **My_alloc<xstring>** to the **My_alloc<char>** that **xstring** needs.

Finally, we can make a **vector** of **xstrings** that uses an allocator of type **My_alloc<xstring>**:

```
svec v {scoped_allocator_adaptor(My_alloc<xstring>{my_arena1})};
```

Now **v** is a **vector** of **strings** using **My_alloc** to allocate memory from **my_arena1** for both **strings** and characters in **strings**.

Why would anyone go to all that bother with allocation? Well, if you have millions of objects and hard real-time requirements on your performance, you can get rather keen on the cost of allocation and deallocation and concerned about the locality of objects. In such cases, having objects and their subobjects in one place can be essential – for example, you may dramatically improve performance by simply “blowing away” a whole allocation arena by a single cheap operation rather than deleting the objects one by one.

Resource management pointers

Resource management is an important part of every non-trivial program. Destructors and the techniques relying on them (notably RAII) are key to most effective resource management strategies. C++0x adds a garbage collection ABI to our toolset, but that must not be seen as a panacea: The question is how to combine destructor-based management of general resources (such as locks, file handles, etc.) with the simple collection of unreferenced objects. This involves non-trivial challenges: for example, destructor-based reasoning is essentially local, scoped, and typed whereas garbage collection is based on non-local reasoning with little use of types (beyond knowing where the pointers are).

Since the mid 1980s, C++ programmers have used counted pointers of various forms to bridge that gap. Indeed, reference-counted objects were the original form of garbage collection (in Lisp) and are still the standard in several languages. They are supported by the standard library **shared_ptr** which provides shared ownership and **weak_ptr** which can be used to address the nasty problems that circular references causes for reference-counted pointers. All **shared_ptr**s for an object share responsibility for the object and the object is deleted when its last **shared_ptr** is destroyed. The simplest example simply provides exception safety:

```
void f(int i)
{
    X* p = new X;
    shared_ptr<X> sp(new X);
    if (i<99) throw Z();    // maybe throw an exception
    delete p;
    // sp's destructor implicitly deletes sp's object
}
```

Here the object pointed to by **p** is leaked if an exception is thrown, but the object pointed to by **sp** is not. However, I am in general suspicious about “shared ownership” which is far too often simply a sign of

weak design and made necessary by a lack of understanding of a system. Consider a common use of a `shared_ptr`:

```

shared_ptr<X> make_X(int i)
{
    // check i, etc.
    return shared_ptr<X>(new X(i));
}

void f(int i)
{
    vector<shared_ptr<X>> v;
    v.push_back(make_X(i));
    v.push_back(make_X(j));
    // ...
}

```

Here we use `shared_ptr` for three things:

- Getting a large object out of a function without copying
- Passing an object from place to place by passing a pointer to it without worrying who eventually needs to destroy it
- Having an owner of the object at all times so that the code is exception-safe (using RAII).

However, we didn't actually share that object in any real sense, we just passed it along in a reasonably efficient and exception-safe manner. The `shared_ptr` implementation keeps a use count to keep track of which is the last `shared_ptr` to an object. If we looked carefully, we'd see that the use count bob up and down between 1 and 2 as the object is passed along before the count finally goes to 0. If we *moved* the pointer around instead of *making copies*, the count would always be 1 until its object finally needed to be destroyed. That is, we didn't need that count! What we saw has been called "false sharing."

C++0x provides a better alternative to `shared_ptr` for the many examples where no true sharing is needed, `unique_ptr`:

- The `unique_ptr` (defined in `<memory>`) provides the semantics of strict ownership.
 - owns the object it holds a pointer to
 - can be moved but not copied
 - stores a pointer to an object and deletes that object when it is itself destroyed (such as when leaving block scope).
- The uses of `unique_ptr` include
 - providing exception safety for dynamically allocated memory,
 - Passing ownership of dynamically allocated memory to a function,
 - returning dynamically allocated memory from a function.
 - storing pointers in containers
- "What `auto_ptr` should have been" (but that we couldn't write in C++98)

Obviously, `unique_ptr` relies critically on rvalue references and move semantics. We can rewrite the `shared_ptr` examples above using `unique_ptr`. For example:

```

unique_ptr<X> make_X(int i)
{
    // check i, etc.
    return unique_ptr<X>(new X(i));
}

void f(int i)
{
    vector<unique_ptr<X>> v;
    v.push_back(make_X(i));
    v.push_back(make_X(j));
    // ...
}

```

The logic is inherently simpler and a **unique_ptr** is represented by a simple built-in pointer and the overhead of using one compared to a built-in pointer are miniscule. In particular, **unique_ptr** does not offer any form of dynamic checking and requires no auxiliary data structures. That can be important in a concurrent system where updating the count for a shared pointer can be relatively costly.

Regular expressions

The absence of a standard regular expression library for C++ has led many to believe that they have to use a “scripting language” to get effective text manipulation. This impression is further enhanced because that a lack of standard also confounds teaching. Since C++0x finally does provide a regular expression library (a derivative of the `boost::regex` library), this is now changing. In a sense it has already changed because I use **regex** to illustrate text manipulation in my new programming textbook [Stroustrup,2008]. I think **regex** is likely to become the most important new library in terms of direct impact on users – the rest of the new library components have more of the flavor of foundation libraries. To give a taste of the style of the **regex** library, let’s define and print a pattern:

```

regex pat (R"[\w{2}s*\d{5}(-\d{4})?]" );    // ZIP code pattern XXdddd-dddd and variants
cout << "pattern: " << pat << '\n';

```

People who have used regular expressions in just about any language will find `\w{2}s*\d{5}(-\d{4})?` familiar. It specifies a pattern starting with two letters `\w{2}` optionally followed by some space `s*` followed by five digits `\d{5}` and optionally followed by a dash and four digits `-\d{4}`. If you have not seen regular expressions before, this may be a good time to learn about them. I can of course recommend my book, but there is no shortage of regular expression tutorials on the web, including the one for `boost::regex` [Maddock,2009].

People who have used regular expressions in C or C++ notice something strange about that pattern: it is not littered with extra backslashes to conform to the usual string literal rules. A string literal preceded by **R** and bracketed by a `[]` pair is a raw string literal. If you prefer, you can of course use a “good old string literal:” `"\\w{2}\\s*\\d{5}(-\\d{4})?"` rather than `R"[\w{2}s*\d{5}(-\d{4})?]"`, but for more complex patterns the escaping can become “quite interesting.” The raw string literals were introduced primarily to

counteract problems experienced with using escape characters in applications with lots of literal strings, such as text processing using regular expressions. The "[...]" bracketing is there to allow plain double quotes in a raw string. If you want a "]" in a raw string you can have that too, but you'll have to look up the detailed rules for raw string bracketing (e.g. in the C++0x FAQ).

The simplest way of using a pattern is to search for it in a stream:

```
int lineno = 0;
string line;    // input buffer
while (getline(in,line)) {
    ++lineno;
    smatch matches;    // matched strings go here
    if (regex_search(line, matches, pat)) // search for pat in line
        cout << lineno << ": " << matches[0] << "\n";
}
```

The `regex_search(line, matches, pat)` searches the `line` for anything that matches the regular expression stored in `pat` and if it finds any matches, it stores them in `matches`. Naturally, if no match was found, `regex_search(line, matches, pat)` returns `false`.

The `matches` variable is of type `smatch`. The “s” stands for “sub”. Basically, a `smatch` is a vector of sub-matches. The first element, here `matches[0]`, is the complete match.

So what does this all add up to?

C++0x feels like a new language – just as C++98 felt like a new language relative to earlier C++. In particular, C++0x does not feel like a new layer of features on top of an older layer or a random collection of tools thrown together in a bag. Why not? It is important to articulate why that is or many might miss something important, just as many were so hung up on OOP that they missed the point of generic programming in C++98. Of course C++ is a general purpose programming language in the sense that it is Turing complete and not restricted to any particular execution environment. But what, specifically, is it good for? Unfortunately, I do not have a snazzy new buzzword that succinctly represents what is unique about C++. However, let me try:

- C++ is a language for building software infrastructure.
- C++ is a language for applications with large systems programming parts.
- C++ is a language for building and using libraries.
- C++ is a language for resource-constrained systems.
- C++ is a language for efficiently expressing abstractions.
- C++ is a language for general and efficient algorithms.
- C++ is a language for general and compact data structures.
- C++ is a lightweight abstraction language.

In particular, it is all of those. In my mind, the first (“building software infrastructure”) points to C++’s unique strengths and the last (“lightweight abstraction”) covers the essential reasons for that.

It is important to find simple, accurate, and comprehensible ways to characterize C++0x. The alternative is to submit to inaccurate and hostile characterizations, often presenting C++ roughly as it was in 1985. But whichever way we describe C++0x, it is still everything C++ ever was – and more. Importantly, I don’t think that “more” is achieved at the cost of greater surface complexity: Generalization and new features can save programmers from heading into “dark corners.”

C++0x is not a proprietary language closely integrated with a huge development and execution infrastructure. Instead, C++ offers a “tool kit” (“building block”) approach that can deliver greater flexibility, superior portability, and a greater range of application areas and platforms. And – of course and essentially – C++ offers stability over decades.

Acknowledgements

The credit for C++0x goes to the people who worked on it. That primarily means the members of WG21. It would not be sensible to list all who contributed here, but have a look at the references in my C++0x FAQ: There, I take care to list names. Thanks to Niels Dekker, Steve Love, Alisdair Meredith, and Roger Orr for finding bugs in early drafts of this paper and to Anthony Williams for saving me from describing threads and locks.

References

[Becker,2009] Pete Becker (editor): *Working Draft, Standard for Programming Language C++*.

[N2914=09-0104] 2009-06-22. Note: The working paper gets revised after each standards meeting.

[Dechev,2009] Damian Dechev and Bjarne Stroustrup: *Reliable and Efficient Concurrent Synchronization for Embedded Real-Time Software*. Proc. 3rd IEEE International Conference on Space Mission Challenges for Information Technology (IEEE SMC-IT). July 2009.

http://www.research.att.com/~bs/smc_it2009.pdf

[Maddock,2009] John Maddock: *Boost.Regex documentation*

http://www.boost.org/doc/libs/1_40_0/libs/regex/doc/html/index.html

[Stroustrup,2009] Bjarne Stroustrup: *C++0x FAQ*. www.research.att.com/~bs/C++0xFAQ.html. Note: this FAQ contains references to the original proposals, thus acknowledging their authors.

[Stroustrup,2002] Bjarne Stroustrup: *Possible Directions of C++0x*. ACCU keynote 2002. Note: No C++0x does not provide all I asked for then, but that’s a different story.

[Stroustrup,2009a] Bjarne Stroustrup: *Concepts and the future of C++* interview by Danny Kalev for DevX. August 2009. <http://www.devx.com/cplus/Article/42448>.

[Stroustrup,2009b] Bjarne Stroustrup: *The C++0x “Remove concepts” Decision*. Dr. Dobbs; July 2009. <http://www.ddj.com/cpp/218600111> and Overload 92; August 2009.

[Stroustrup,1994] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. 1994.

[Stroustrup,2008] Bjarne Stroustrup: *Programming: Principles and Practice using C++*. Addison-Wesley. 2008.

[Williams,2009] Anthony Williams: *Multi-threading in C++0x*. Overload 93; October 2009.