# C++ in 2005

"Living languages must change,
must adapt,
must grow."
– Edward Finegan

This extended foreword presents a perspective on "The Design and Evolution of C++" and on C++ itself. In particular, it reflects on the use of C++ over the last decade and presents plausible directions for the next revision of the ISO C++ standard, C++0x.

The central parts tell the success story of the STL ("Standard Template Library"), of the near disaster of the separate compilation of templates ("export") discussion, and of the definition of "exception safety". The technical reports on performance and libraries are also presented. The section on the future focuses on the likely language extensions to better support generic programming: concepts for better template argument checking and separate compilation of templates and a generalization of initializer lists primarily in support for more elegant use of containers.

The general organization of this "extended foreword" is

1. "The Design and Evolution of C++"
2. Where we are – 2005
3. Where we were – 1995-2004
4. Where we might be going – 2005-2014

Obviously, the 2005-2014 section contains mostly conjecture.

## 1 "The Design and Evolution of C++"

"The Design and Evolution of C++", often called D&E, is the personal favorite among my books. Writing it, I was free of the usual rigid constraints of textbook and academic paper styles. There was essentially no precedent for writing a book retrospectively about the design of a language, so I could simply tell the story about how C++ came about, why it looks the way it does, and why alternative decisions were not taken. I could thank my friends and colleagues for their contributions and even state where I thought I had made mistakes. It was fun to write and re-reading it, I think that pleasure of writing shines through to make it a more pleasant read than much conventional technical prose.

The book has aged beautifully. From a programming language point of view, hardly anything has happened to C++ over the last decade. What we have gained is an immense amount of experience with its use, so that I can now speak with far greater confidence than was possible in 1994. Amazingly, "The Design and Evolution" is still the most complete and up-to-date treatment of its topic. That's partly because history and design decisions don't change and partly because the vast C++ literature is focused on the use of C++ rather than the reasons behind its design.

Some have dismissively called D&E "just a history book". That's inaccurate. Understanding why C++ is the way it is helps a programmer use it well. A deep understanding of a tool is essential for an expert craftsman. That's why D&E received Software Development's "Productivity Award". History books do not get awards for increasing productivity. D&E has even been used as an introduction to C++ for people who appreciate concepts, wants to know about language implementation models, and learn from examples. It is most definitely not a book aimed at teaching programming techniques.

D&E was completed just before the feature set for the ISO C++ standard was "frozen" and a minor revision for the 3rd printing in 1995 (which is the source of this translation) corrected a handful of small errors relative to the standard. I am most happy to report that the majority of these corrections were of the form of a change from "I hope that X" to "The standard says X".

Some readers have found D&E lacking in idealism. That surprised me, but I guess it shouldn't have. Some want to see a programming language as nothing but a beautiful piece of logic sprung complete from the head(s) of its genius designer(s). You certainly won't find that here. I tell the story of C++, warts and all. The aims of C++ are noble: enable programmers to write real-world programs that are simultaneously elegant and efficient, to raise the level of abstraction in real-world code, and thereby improve the working lives of hundreds of thousands of serious programmers. The snag is the adjectives "real-world" and "serious". I really don't worry too much about toy problems isolated from code written to deliver some service to its users. Once you place real-world constraints on code, absolute beauty becomes hard to apply as a principle and engineering tradeoffs become necessary. People who don't accept that can't help having a problem with the design and evolution of C++.

# 2 Where we are – 2005

As ever, it is hard to estimate the number of C++ programmers, but in 2003, the IDC reported well over three million full-time C++ programmers (compared to my estimate of 400,000 in 1991; §D&E7.1), and that's not an implausible number. I'm not in a position to accurately count, but all the indicators I have, show a steady increase in the use of C++ over the last decade (1995-2004) after the explosive growth of C++'s first decade (1985-1994). I never experienced a year without growth.

My guess is that one of the main reasons for this steady growth – in the face of vigorous promotion and occasionally unscrupulous marketing of alternatives – is exactly the stability of the C++ language over that time period. The implementations of the language have improved immensely over the years, the libraries have grown in number, quality, and sophistication, and our understanding of how to use C++ is far deeper today. However, the language described in D&E in 1995 is the language we use today. Standard C++ has enough features to accommodate the growth in programming techniques and applications over the decade and the stability has allowed the implementers to catch up.

So what do all of those C++ programmers actually do? What kind of applications do they write and what kind of programming styles do they employ? I don't know; nobody knows. In the same way as there are too many C++ programmers to count, there are too many different application areas and too many programming styles for any one person to grasp. It is common to hear generalizations along the line "C++ is used like this". Such statements are typically wishful thinking based on very limited experience. We are playing "blind men and the elephant" with a very large creature. There are people who have read more than a million lines of C++ code, written hundred of thousands of lines of C++, read all the articles in C-vu, C/C++ Users Journal, etc., read all the good C++ books and dozens of the bad ones, read all the academic papers relating to C++, and "lived" on the C++ newsgroups for years. There are not many such people, and even they have only scratched the surface. Such people are usually the last to utter simple generalizations. In fact, I hear the most succinct and confident generalizations (both positive and negative) about C++ from people who have hardly any experience with C++. Ignorance is bliss.

When I try to think about how C++ is used, I first consider two dimensions:

- Application area
- Maturity of programmers (designers, software producing organizations, etc.)

Programmers writing hard-real time (embedded) systems really do have different concerns from programmers of database-limited business programs, and both live in a completely different world from the programmers of high-energy physics applications. I always find it instructive to listen to programmers from a new application area and to learn from them.

It is very hard to say something that makes sense across all application areas. However, it is possible to say something about maturity. From a high-level perspective, the ideals of a programming language can be expressed as

1. express concepts directly in code
2. express relations among concepts directly in code
3. express independent concepts in independent code
4. compose code representing concepts freely wherever the composition makes sense

Here “concept” corresponds roughly to “idea” and refers to anything we name, appears on our blackboard when we design, are described in our textbooks, etc.

I “measure” maturity primarily based on how close people get to those ideals in production code (i.e., in code suffering real-world constraints). People who use C++ primarily as “a better C” fail on the first count – they fail to use classes, class hierarchies, and parameterization (templates) to express ideas and relations among ideas directly. People who insist seeing C++ as just an object-oriented language fail on the third and fourth – they construct massive hierarchies that bind unrelated concepts together through unsuitable bases and exclude built-in types and simple classes.

This means that there is a vast scope for improvement without further language changes. Most people can improve their programming productivity, decrease their error rate, and improve run-time performance simply by using the tools already on their machine as part of ISO Standard C++. If you haven’t tried the STL, that would be a good place to start (see §3.1). It may not be exactly what you need, but it’s standard and demonstrates many of the key techniques of “modern C++” that you can apply to your own problems. The education problem that I point out in §D&E9.4.2 is even worse today: To an astonishing degree, the teaching of programming has failed to keep up with the changes in the way software is produced. Since I originally wrote D&E, I have become a professor, partially to help reverse that trend.

Again, so what do all of those C++ programmers actually do? Just about anything you can think of: “ordinary PC business applications”, embedded systems, e-commerce, games, scientific computation, network software, operating systems, device drivers, cell phones, etc. Instead of going on forever, I suggest you have a look at a little list I have been maintaining: http://www.research.att.com/~bs/applications.html. Personally, I take special pleasure in “adventurous and unusual” applications with extreme performance and/or reliability requirements such as the JPL Mars Rover autonomous driving system, the MAN B&W control system for huge marine diesel engines, and the ICE infrastructure for highly distributed systems (such as multi-player games).

At this point, I suggest you proceed with the original foreword and the first chapters only to return here when you have completed Part I. The next section here continues the C++ story where Chapter 9 leaves off.

# 3 Where we were – 1995-2004

In 1994, my primary concern was to get the ISO C++ standard as good as possible – both in terms of features included and the quality of the specification – and to gain a consensus for it. It does not matter how good a specification is if people don’t accept it. There is no enforcement of an ISO standard, so if someone decides that it is not worth their while to conform only community pressure can convince them otherwise. For an implementer, conforming is significant extra work, so conforming takes a conscious decision and allocation of resources potentially spent elsewhere. There are obscure language features that can be hard to implement in some compilers. There are libraries to implement or buy, and there are opportunities to lock in users with fancy proprietary features that a responsible implementer must forego. Thus, I considered it essential that the members of the committee and the organizations they represented actually believed the standards document to be the very best they could hope for.

After much work the committee succeeded. The final vote among the technical members at the Morristown (New Jersey, USA) meeting in October 1997 was 43-0. We celebrated suitably after that! In 1998, the ISO member nations ratified the standard with an unprecedented 22-0 vote. Achieving that consensus took a lot of technical work and some diplomacy: At the time, I was fond of saying "political problems cannot be solved; we must find the technical problem that is the real cause of the problem and solve that". I can't think of a single major problem that was "solved" by simply voting down the minority, and only one issue where "political horse trading" compromised our best technical judgment – and that problem (separate compilation of templates) still festers, looking for a better technical solution.

In the years before the final votes, three things dominated the committee's work:

1. Details, details, and more details
2. The STL
3. Separate compilation of templates

The first is obvious: An international standard must pay a great deal of attention to details; after all, conformance of implementations to the written standard is the key aim of a standard and the basis for portability of applications and tools between implementations. The standard is a 712 page (plus index, etc.) document [ISO, 1998] written in a highly technical and formal style, so there are a lot of details to get right. As before, I followed up the new language specification with a new edition of "The C++ Programming Language" [Stroustrup, 1998] to provide a more tutorial and user-oriented description of the language.

## 3.1 The emergence of the STL

The second issue, the STL (the "Standard Template Library"; that is, the containers and algorithm framework of the ISO C++ standard library), turned out to be the major innovation to become part of the standard and the starting point for much of the new thinking about programming techniques that have occurred since. Basically, the STL was a revolutionary departure from the way we had been thinking about containers and their use. From the earliest days of Simula, containers (such as lists) had been intrusive: An object could be put into a container if and only if its class had been (explicitly or implicitly) derived from a specific "Link" or "Object" class containing the link information needed by the compiler. Basically, such a container is a container of references to Links. This implies that fundamental types, such as **int**s and **double**s, can't be put directly into containers and that the array type, which directly supports fundamental types, must be different from other containers. Furthermore, objects of really simple classes, such as **complex** and **Point**, can't remain optimal in time and space if we want to put them into a container. It also implies that such containers are not statically type safe. For example, a **Circle** may be added to a list, but when it is extracted we know only that it is an **Object** and need to apply a cast (explicit type conversion) to regain the static type.

Simula containers and arrays had this irregular treatment of built-in and user-defined types (only some of the latter could be in containers) and of containers and arrays (only arrays could hold fundamental types; arrays couldn't hold user-defined types, only references to user-defined types). Smalltalk has the same approach and the same problems, as have later languages such as Java and C#. Many C++ libraries also followed this model because it does have significant utility and many designers are by now familiar with it. However, I had found this irregularity and the inefficiency (in time and space) that goes with it unacceptable for a truly general-purpose library (you can find a summary of my analysis in §16.2 of [Stroustrup,2000]). This was the fundamental reason behind my mistake of not providing a suitable standard library from C++ in 1985 (see §D&E9.2.3).

When I wrote D&E, I had become aware of a new approach to containers and their use, which had been developed by Alex Stepanov. Alex then worked at HP Labs but he had earlier worked for a couple of years at Bell Labs, where he had been close to Andrew Koenig and where I had discussed library design and template mechanisms with him. He had inspired me to work harder on generality and efficiency of some of the template mechanisms, but fortunately he failed to convince me to make templates more like Ada generics. Had he succeeded, he wouldn't have been able to design and implement the STL!

In late 1993, Alex showed the latest development in his decades long research into generic programming techniques aiming for "the most general and most efficient" programming techniques based on a rigorous mathematical foundation. It was a framework of containers and algorithms. He first contacted Andrew, who after playing with it for a couple of days showed it to me. My first reaction was puzzlement. I found the STL style of containers and container use very odd, even somewhat ugly. Like many programmers acquainted with object-oriented programming, I thought I knew how containers had to look and STL code looked very different. However, over the years I had developed a checklist of properties that I considered important for containers and to my amazement the STL met all but one of the criteria on that list! The missing criterion was the use of the common base class to provide services (such as persistence) for all derived classes (e.g., all objects or all containers). However, I didn't (and don't) consider such services intrinsic to the notion of a container.

It took me some time – weeks – to get comfortable with the STL. After that, I worried that it was too late to introduce a completely new style of library into the C++ community. Looking at the odds to get the standards committee to accept something new and revolutionary at such a late stage of the standards process, I decided (correctly) that those odds were very low. Even at best, the standard would be delayed by a year – and the C++ community badly needed that standard. Also, the committee is fundamentally a conservative body and the STL was revolutionary.

So, the odds were poor, but I plodded on hoping. After all, I really did feel very bad about C++ not having a sufficiently large and sufficiently good standard library (§D&E9.2.3). Andrew Koenig did his best to build up my courage and Alex Stepanov lobbied Andy and me as best he knew. Fortunately, Alex didn't quite understand the difficulties of getting something major though the committee, so he was less daunted and worked on the technical aspects and on teaching Andrew and me. I began to explain the ideas behind the STL to others; for example, the examples in §D&E15.6.3.1 came from the STL and the quote by Alex Stepanov in §D&E15.11.2 is about the STL.

We invited Alex to give an evening presentation at the October 1993 standard committee meeting in San Jose, California: "It was entitled *The Science of C++ Programming* and dealt mostly with axioms of regular types – connecting constructions, assignment and equality. I did also described axioms of what is now called Forward Iterators. I did not at all mention any containers and only one algorithm: find." [Stepanov,2004]. That talk was an audacious piece of rabble rousing that to my amazement and great pleasure basically swung the committee away from the attitude of "its impossible to do something major at this stage" to "well, let's have a look".

That was the break we needed! Over the next four month we experimented, argued, lobbied, taught, programmed, and redesigned so that Alex was able to present a complete description of the STL to the committee at the March 1994 meeting in San Diego, California. At a meeting, that Alex arranged for C++ library implementers at HP later in 1994, we agreed on many details, but the size of the STL emerged as the major obstacle. Finally, at Alex's urging, I took a pen and literally crossed out something like two thirds of all the text. For each facility, I challenged Alex and the other library experts to explain – very briefly – why it couldn't be cut and why it would benefit most C++ programmers. It was a horrendous exercise. Alex later claimed that it broke his heart. However, what emerged from that slashing is what is now known as the STL [Stepanov, 1994] and it made it into the ISO C++ standard at the October 1994 meeting in Waterloo, Canada – something that the original and complete STL would never had done. Even the necessary revisions of the "reduced STL" delayed the standard by more than a year. In retrospect, I think that I did less damage than we had any right to hope for.

Among all the discussions about the possible adoption of the STL one memory stands out: Beman Dawes calmly stating that he had thought the STL too complex for ordinary programmers, but as an exercise he had implemented about 10% of it himself so he no longer considered it beyond the standard. Beman was one of the all too rare application builders in the committee. Unfortunately, the committee tends to be dominated by compiler, library, and tools builders.

I credit Alex Stepanov with the STL. He worked with the fundamental ideals and techniques for well over a decade before the STL, unsuccessfully using languages such as Scheme and Ada. However, Alex would be the first to insist that others took part in that quest. David Musser has been working with Alex on generic programming for almost two decades and Meng Lee worked closely with him at HP helping to program the original

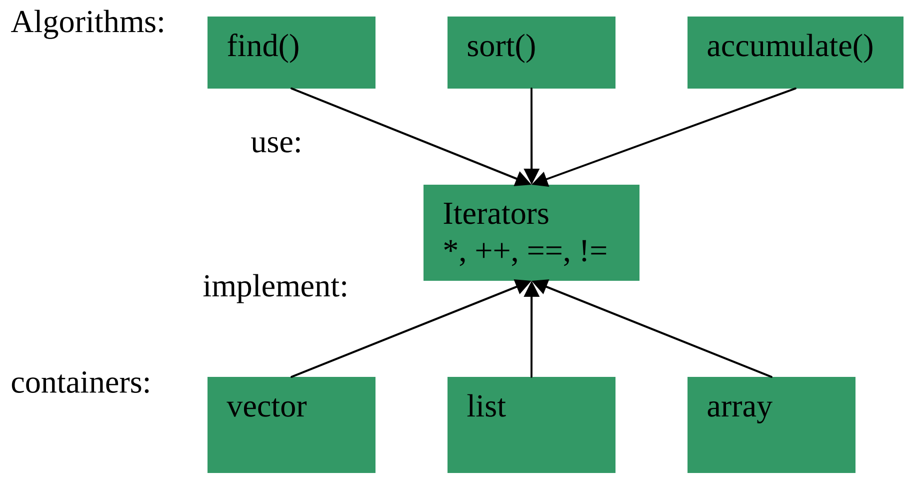
STL. Email discussions between Alex and Andrew Koenig also helped. Apart from the slashing exercise, my contributions were minor. I suggested that various information related to memory be collected into a single object – what became the allocators. I also drew up the initial requirements tables on Alex's blackboard, thus creating the form in which the requirements of the STL algorithms and classes on their template arguments are documented. These requirements tables are actually an indicator that the language is insufficiently expressive – such requirements should be part of the code; see §4.1.

### 3.1.1 STL ideals and concepts

So what is the STL? It has now been part of Standard C++ for almost a decade, so you really should know, but if you are new to modern C++ here is a brief explanation with a bias towards ideals and language usage.

Consider the problem of storing objects in containers and writing algorithms to manipulate such objects. Consider this problem in the light of the ideals of direct, independent, and composable representation of concepts listed in §2. Naturally, we want to be able to store objects of a variety of types (e.g. **int**s, **Point**s, pointers to **Shape**s) in a variety of containers (e.g. **list**s, **vector**s, **map**s) and to apply a variety of algorithms (e.g. **sort**, **find**, **accumulate**) to the objects in the containers. Furthermore, we want the use of these objects, containers, and algorithms to be statically type safe, as fast as possible, as compact as possible, not verbose, and readable. Achieving all of this simultaneously isn't easy. In fact, I spent about 10 years unsuccessfully looking for a solution to this puzzle.

The STL solution is based on parameterizing containers with their element types and on completely separating the algorithms from the containers. Each type of container provides an iterator type and all access to the elements of the container can be done using only those iterators. That way, an algorithm can be written to use iterators without having to know about the container that supplied them. Each type of iterator is completely independent of all others except for supplying the same semantics to required operations, such as ***** and **++**. We can illustrate this graphically:



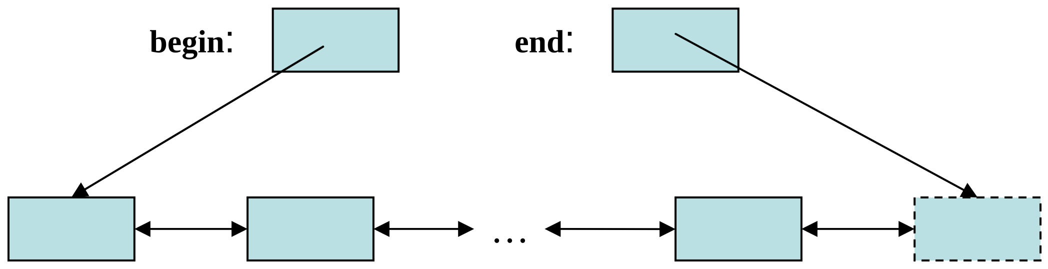
Let's consider a fairly well-known example based on finding elements of various types in various containers. First, here are a couple of containers

```
vector<int> vi;        // vector of ints
list<double> vd;       // list of doubles
```

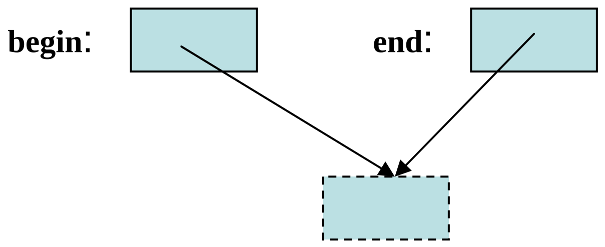
These are the standard library versions of the notions of vector and list implemented as templates (§D&E15.3). Assume that they have been suitably initialized with values of their respective element types. It then makes sense to try to find the first element with the value **7** in **vi** and the first element with the value **3.14** in **vd**:

```
vector<int>::iterator  p = find(vi.begin(),vi.end(),7);
list<double>::iterator q = find(vd.begin(),vd.end(),3.14);
```

The basic idea is that you can consider the elements of any container as a sequence of elements. A container "knows" where its first element is and where its last element is. We call an object that points to an element "an iterator". We can then represent the elements of a container by a pair of iterators, **begin()** and **end()**, where **begin()** points to the first element and **end()** to one-beyond-the-last element:



The end() iterator points to one-past-the-last element rather than to the last element to allow the empty sequence not to be a special case:



What can you do with an iterator? You can get the value of the element pointed to (using ***** just like for a pointer), make the iterator point to the next element (using **++** just like for a pointer) and compare two iterators to see if they point to the same element (using **==** or **!=** of course). Surprisingly, this is sufficient for implementing **find()**:

```
template<class Iter, class T> Iter find(Iter first, Iter last, const T& val)
{
        while (first!=last && *first!=val) ++first;
        return first;
}
```

This is a simple – very simple really – template function as described in §D&E15.6. People familiar with C and C++ pointers should find the code easy the read: **first!=last** checks whether we have reached the end and ***first!=val** checks whether we have found the value **val** that we were looking for. If not, we increment the iterator **first** to make it point to the next element and try again. Thus, when **find()** returns its value will point to either the first element with the value **val** or one-past-the-last element (**end()**). So we can write:

```
vector<int>::iterator p = find(vi.begin(),vi.end(),7);

if (p != vi.end()) {        // we found 7
        // …
}
else {                      // no 7 in vi
        // …
}
```

This is very, very simple. It is simple like the first couple of pages in a Math book and simple enough to be really fast. However, I know that I'm not the only person to take significant time figuring out what really is going on here and longer to figure out why this is actually a good idea. Like simple Math, the first STL rules and principles generalize beyond belief.

Consider first the implementation: In the call **find(vi.begin(),vi.end(),7)**, the iterators **vi.begin()** and **vi.end()** that become **first** and **last**, respectively, inside **find()** are something that points to an **int**. The obvious implementation of **vector<int>::iterator** is therefore a pointer to **int**, an **int***. With that implementation, * becomes pointer dereference ++ becomes pointer increment, and != becomes pointer comparison. That is, the implementation of **find()** is obvious and optimal. In particular, we do not use function calls to access the operations (such as ***** and **!=**) that are effectively arguments to the algorithm because they depend on a template argument. In this, templates differ radically from most mechanisms for "generics", relying on indirect function calls (like virtual functions), in current programming languages. Given a good optimizer, **vector<int>::iterator** can without overhead be a class with * and ++ provided as inline functions. Such optimizers are now not uncommon and using a class improves type checking by catching unwarranted assumption, such as

```
int* p = find(vi.begin(),vi.end(),7);  // oops: the iterator type need not be int*
```

So why didn't we just dispense with all that "iterator stuff" and use pointers? One reason is that **vector<int>::iterator** could have been a class providing range checked access. For a less subtle explanation, have a look at the other call of **find()**:

```
list<double>::iterator  q= find(vd.begin(),vd.end(),3.14);

if (q != vd.end()) {     // we found 3.14
        // …
}
else {                   // no 3.14 in vi
        // …
}
```

Here, **list<double>::iterator** isn't going to be a **double***. In fact, assuming the most common implementation of a linked list, **list<double>::iterator** is going to be a **Link*** where **Link** is a link node type, such as:

```
template<class T> struct Link {
        T value;
        Link* suc;
        Link* pre;
};
```

That means that * means **p->value** ("return the value field"), ++ means **p->suc** ("return a pointer to the next link"), and != pointer comparison (comparing **Link***s). Again the implementation is obvious and optimal. However, it is completely different from what we saw for **vector<int>::iterator**.

We have used a combination of templates and overload resolution to pick radically different, yet optimal, code for a single source code definition **find()** and for the uses of **find()**. Note that there is no run-time dispatch, no virtual function calls. In fact, there are only calls of trivially inlined functions and fundamental operations such as ***** and **++** for a pointer. In terms of speed and code size, we have hit rock bottom!

Why not use "sequence" or "container" as the fundamental notion rather than "pair of iterators"? Part of the reason is that "pair of iterators" is simply a more general concept than "container". For example, given iterators, we can sort the first half of a container only: **sort(vi.begin(), vi.begin()+vi.size()/2)**. Another reason is that the STL follows the C++ design rules that we must provide transition paths and support-built in and user-defined types uniformly. What if someone kept data in an ordinary array? We can still use the STL algorithms. For example:

```
char buf[max];
// … fill buf …
int*  p = find(buf,buf+max,7);

if (p != buf+max) {    // we found 7
        // …
}
else {                 // no 7 in buf
        // …
}
```

Here, the *****, **++**, and **!=** in **find()** really are pointer operations! Like C++ itself, the STL is compatible with older notions such as C arrays. As always, we provide a transition path (§D&E4.2). This also serves the ideal of providing uniform treatment to user-defined types (such as **vector**) and built-in types (in this case, array) (§D&E4.4).

The STL, as adopted as the containers and algorithms framework of the ISO C++ standard library, consists of a dozen containers (such as **vector**, **list**, and **map**) and data structures (such as arrays) that can be used as sequences. In addition, there are about 60 algorithms (such as **find**, **sort**, **accumulate**, and **merge**). It would not be reasonable to present all of those here. For details, see [Austern, 1998], [Stroustrup, 2000].

The key to both the elegance and the performance of the STL is that it – like C++ itself (§D&E2.5.2, §D&E2.9.3, §D&E3.5.1, §D&E12.4) – is based directly on the hardware model of memory and computation. The STL notion of a sequence is basically that of the hardware's view of memory as a set of sequences of objects. The basic semantics of the STL maps directly into hardware instructions allowing algorithms to be implemented optimally. The compile-time resolution of templates and the perfect inlining they support is then key to the efficient mapping of high level expression of the STL to the hardware level.

### 3.1.2 Function objects

I'd like to present one more of the essential techniques used by the STL to show how it builds on general C++ mechanisms to provide unprecedented flexibility and performance. The STL framework, as described so far, is somewhat rigid. Each algorithm does exactly one thing in exactly the way the standard specifies it. For example, we find an element that is equal to the value we specify as an argument. It is actually more common to look for an element that has some desired property, such as being less than a given value or matching a value given a criterion different from simple equality (e.g., matching strings without case sensitivities or matching double values allowing for very slight differences).

As an example, instead of finding a value **7**, let's look for a value that meets some predicate, say, being less than **7**:

```
vector<int>::iterator p = find_if(v.begin(),v.end(),Less_than<int>(7));

if (p != vi.end()) {        // we found an element with a value < 7
        // …
}
else {                      // no element with a value < 7 in vi
        // …
}
```

What is **Less_than<int>(7)**? It is a function object, that is an object of a class that has the application operator, **( )**, defined to perform a function:

```
template<class T> struct Less_than {
        T value;
        Less_than(const T& v) :value(v) { }
        bool operator()(const T& v) const { return v<value; }
};
```

For example:

```
Less_than<double> f(3.14); // Less_than object holding the double 3.14
bool b1 = f(3);            // b1 becomes true (3<3.14 is true)
bool b2 = f(4);            // b2 becomes false (4<3.14 is false)
```

From the vantage point of 2004, it seems odd that function objects are not mentioned in D&E. They deserve a whole section. Even the use of a user-defined application operator, **( )**, isn't mentioned even though it has had long and distinguished career. For example, it was among the initial set of operators (after =; see §D&E3.6) that I allowed to be overloaded and was among many other things used to mimic Fortran subscript notation.

We can write a version of **find()** that uses a function object, rather than plain **!=** to determine whether an element is to be found or not. It is called **find_if()**:

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
        while (first!=last && !pred(*first)) ++first;
        return first;
}
```

We simply replaced ***first!=val** with **!pred(*first)**. The function template **find_if()** will accept any object that can be called given an element value as its argument. In particular, we could call **find_if()** with an ordinary function as its third argument:

```
bool less_than_7(int a)
{
        return 7<a;
}

vector<int>::iterator p = find_if(v.begin(),v.end(),less_than_7);
```

However, this example shows why we often prefer a function object over a function: The function object can be initialized with one (or more) argument and carry that information along for later use. A function object can carry a state. That makes for more general and more elegant code. If needed, we can also examine that state later. For example:

```
template<class T> struct Accumulator {    // keep the sum of n values
        T value;
        int count;
        Accumulator() :value(), count(0) { }
        Accumulator(const T& v) :value(v), count(0) { }
        void operator()(const T& v) { ++count; value+=v; }
};
```

An **Accumulator** object can be passed to an algorithm that calls it repeatedly. The partial result is carried along in the object. For example:

```
int main()
{
        vector<double> v;
        double d;
        while (cin>>d) v.push_back(d);

        Accumulator<double> ad;
        ad = for_each(v.begin(),v.end(), ad);
        cout << "sum==" << ad.value
              << ", mean==" << ad.value/ad.count << '\n';
}
```

The standard library algorithm **for_each** simply applies its third argument to each element of its sequence and returns that argument as its return value. The alternative to using a function object would be a messy use of global variables to hold **value** and **count**.

Interestingly, simple function objects tend to perform better than their function equivalents. The reason is that they tend to be passed by value so that they are easier to inline. This can be very significant when we pass an object or function that performs a really simple operation, such as the comparison criteria for a sort. In particular, inlining of function objects is the reason that the STL (C++ standard library) **sort()** to outperforms the conventional **qsort()** by several factors when sorting arrays of types with simple comparison operators (such as **int** and **double**) [Stroustrup,1999].

Function objects are the C++ mechanism for higher-order constructs. It is not the most elegant expression of high-order ideas, but it is surprising expressive and inherently efficient in the context of a general purpose language. As an example of expressiveness, Jaakko Järvi showed how to provide and use a lambda class that made this legal with its obvious meaning:

```
Lambda x;
List<int>::iterator p = find_if(lst.begin(),lst.end(),x<=7);
```

If you want just **<=** to work, rather than a building a general library, you can add definitions for **Lambda** and <= in less than a dozen lines of code. Using **Less_than** from the example above, we can simply write:

```
class Lambda {};

template<class T> Less_than<T> operator<=(Lambda,const T& v)
{
        return Less_than<T>(v);
}
```

So, the argument **x<=7** in the call of **find_if** becomes a call of **operator<=(Lambda,const int&)** which generates a **Less_than<int>** object just as we used in the first example in this section. The difference here is simply that we have achieved a much simpler and more intuitive syntax. This is a good example of the expressive power of C++ and of how the interface to a library can be simpler than its implementation. Naturally, there is no run-time overhead compared to a laboriously written loop to look for an element with a value less than **7**.

### 3.1.3 The impact of the STL

The impact of the STL on the thinking of C++ has been immense. Before the STL, I consistently list three fundamental programming styles (“paradigms”) as being supported by C++ (§D&E4.1):

- Procedural programming
- Data abstraction
- Object-oriented programming

I considered templates as support for data abstraction. After playing with the STL for a while, I factored out a fourth style

– Generic programming

The techniques based on the use of templates and largely inspired by techniques from functional programming are qualitatively different from traditional data abstraction. People simply think differently about types, objects, and resources. New C++ libraries are written – using templates – to be statically type safe and efficient. Templates are the key to embedded systems programming and high-performance numeric programming where resources management and correctness are key. The STL itself is not always ideal in those areas. For example, it doesn't provide direct support for linear algebra and it can be tricky to use in hard-real time systems where free store use is typically banned. However, the STL demonstrates what can be done with templates and gives examples of effective techniques. For example, the use of iterators (and allocators) to separate logical memory access from actual memory access is key to many high-performance numeric techniques and the use of small, easily inlined, objects are key to examples of optimal use of hardware in embedded systems programming. Some of these techniques are documented in the standard committee's technical report on performance [ISO, 2004]. This is to a large extent a reaction to – and a constructive alternative to – a trend towards overuse of "object oriented" techniques relying excessively on class hierarchies and virtual functions.

Obviously, the STL isn't perfect. There is no one "thing" to be perfect relative to. However, it broke new ground and has had impact even beyond the huge C++ community. Using C++, people talk about "template meta-programming" when they try to push the techniques pioneered by the STL beyond the STL. Some of us also think about the limits of STL iterators (where would it be preferable to use generators and ranges?) and about how C++ could better support these uses (concepts, initializers, see §4).

## 3.2 The export controversy

The other major story of the later years of the standards effort is less uplifting. In fact, it almost ended in tragedy and left many members of the committee (me included) unhappy for years. From the earliest designs, templates were intended to allow a template to be used after specifying just a declaration and not a definition in a translation unit. It is then the job of the implementation to find and use the definition of the template appropriately (§D&E15.10.4). That's the way it is for other language constructs, such as functions, but for templates that's easily said but extremely hard to do.

In 1996, a vigorous debate erupted in the committee over whether we should not just accept the "include everything" model for template definitions, but actually outlaw the original model of separation of template declarations and definitions into separate translation units. The arguments of the two sides were basically

- Separate translation of templates is too hard (if not impossible) and such a burden should not be imposed on implementers

- Separate translation of templates is necessary for proper code organization (according to data hiding principles)

Many subsidiary arguments supported both sides. I was on the side that insisted on separate compilation of templates. As ever in really nasty discussions, both sides were mostly correct on key points. In the end, people from SGI – notably John Wilkinson – proposed a new model that was accepted as a compromise. The compromise was named after the keyword used to indicate that a template could be separately translated: **export**.

The issues fester to this day: as late as 2003, Herb Sutter and Tom Plum proposed a change to the standard declaring an implementation that didn't implement separate compilation of templates conforming. The reason given was again implementation complexity plus the fact that even five years after the standard was ratified only one implementation existed. That motion was defeated by a 80/20 vote, partly because an implementation of **export** now exists.

The real heroes of this sad tale are the implementers of the EDG compiler: Steve Adamczyk, John Spicer, and David Vandevoorde. They strongly opposed separate compilation of templates, finally voted for the standard as the best compromise attainable, and then proceeded to spend more than a year implementing what they had opposed. That's professionalism! The implementation was every bit as difficult as its opponents had predicted but it worked and actually provided some (but not all) of the benefits that its proponents had promised. Unfortunately, some of the restrictions on separately compiled templates that proved essential for a compromise ended up not providing their expected benefits and complicated the implementation. As ever, political compromises on technical issues led to "warts".

## 3.3 Exception Safety

During the effort to specify the STL we encountered a curious phenomenon: We didn't quite know how to talk about the interaction between templates and exceptions. Quite a few people were placing blame for this problem on templates and others began to consider exceptions fundamentally flawed [Carghill, 1994] or at least fundamentally flawed in the absence of automatic garbage collection. However, when a group of "library people" (notably Matt Austern, Greg Colvin, and Dave Abrahams) looked into this problem, they found that we basically had a language feature – exceptions – that we didn't know how to use well. The problem was in the interaction between resources and exceptions. I had of course considered this when I designed the exception handling mechanisms and come up with the rules for exceptions thrown from constructors (correctly handling partially constructed composite objects) and the "resource acquisition is initialization" technique (§D&E16.5). However that was only a good start and an essential foundation. What we needed was a conceptual framework – a more systematic way of thinking about resource management.

Dave Abrahams condensed the result of work over a couple of years in three concepts:

- The *basic* guarantee: that the invariants of the component are preserved, and no resources are leaked.
- The *strong* guarantee: that the operation has either completed successfully or thrown an exception, leaving the program state exactly as it was before the operation started.
- The *no-throw* guarantee: that the operation will not throw an exception.

Using these fundamental concepts, the library working group described the standard library and implementers produced efficient and robust implementations. The standard library guarantees the basic guarantee for all operations with the caveat that no destructor may exit through an exception throw. In addition, the library provides the strong guarantee and the no-throw guarantee for key operations. I found this result important enough to add a chapter to [Stroustrup, 1998] yielding [Stroustrup, 2000]. For details of the standard library exception guarantees and programming techniques for using exceptions, see Appendix E of [Stroustrup, 2000].

I think the key lesson here is that it is not sufficient just to know how a language feature behaves. To write good software, we must have a clearly articulated design strategy for problems that require the use of the feature.

## 3.4 Maintaining the standard

After a standard is passed, the ISO process goes into a “maintenance mode” for at least five years. During that time the committee primarily responds to reports of defects. Most defects are resolved by clarifying the text or resolving contradictions. Only very rarely will new rules be introduced and real innovation is avoided. Stability is the aim. In 2003, all these minor correction were published under the name “Technical Corrigenda 1”. At the same time, members of the British national committee took the opportunity to remedy a long-standing problem: they got Wiley to publish a printed version of the (revised) standard [ISO, 2003b]. The initiative and much of the hard work came from Francis Glassborow and Lois Goldthwaite with technical support from the committee’s project editor, Andrew Koenig, who produced the actual text.

Until the publication of the revised standard in 2003, the only copies of the standard available to the public were a very expensive (about $200) paper copy from ISO or ANSI or a cheap ($18) pdf version from ANSI. The pdf version was a complete novelty at the time. Standards bodies are partially financed through the sales of standards, so they are most reluctant to make standards available for free or cheaply. In addition, they don’t have retail sales channels, so you can’t find a national or international standard in your local book store – except the C++ standard, of course. Following the C++ initiative, the C standard is now also available.

Maintenance wasn’t all that the committee did from 1997 to 2003. There was a modest amount of planning for the future (thinking about C++0x), but the main activities was the writing a technical report on performance issues [ISO, 2004] and one on libraries [ISO, 2005].

## 3.4.1 The Performance TR

The Performance TR was prompted by a suggestion to standardize a subset of C++ for embedded systems programming. The proposal, called Embedded C++ [EC++, 1999] or simply EC++, originated from a consortium of Japanese embedded systems tool developers and had two main concerns: removal of language features that potentially hurt performance and removal of language features perceived to be too complicated for programmers (and thus seen as potential productivity or correctness hazards). A largely unstated aim was to define something that was easier to implement than full standard C++.

The features banned in this (almost) subset included: multiple inheritance, templates, exceptions, run-time type information, new-style casts, and name spaces. From the standard library, the STL and locales were banned and an alternative version of **iostream**s provided. I considered the proposal misguided and backwards looking. In particular, the performance costs were largely imaginary, or worse (i.e., a feature absent from EC++ was essential for performance in some domain). For example, the use of templates has repeatedly been shown to be key to both performance (time and space) and correctness of embedded systems. However, there wasn't much hard data in this area in 1996 when EC++ was first proposed. Ironically, it appears that most of the few people, who use EC++ today, use it in the form of Extended EC++, which is EC++ plus templates. Similarly, namespaces and new style casts are primarily features that are primarily there to clarify code and can be used to ease maintenance and verification of correctness.

After serious consideration and discussion, the ISO committee decided to stick to the long-standing tradition of not endorsing dialects – even dialects which are (almost) subsets. Every dialect leads to a split in the user community, and so does even a formally defined subset when its users start to develop a separate culture of techniques, libraries, and tools. Inevitably, myths about failings of the full language relative to the "subset" will start to emerge. Thus, I recommend against the use of EC++ in favor of using what is appropriate from (full) ISO Standard C++.

Obviously, the EC++ proposers were right in wanting an efficient, well-implemented, and relatively easy-to-use language. It was up to the committee to demonstrate that Standard C++ was that language. In particular, it seemed a proper task for the committee to document the utility of the features rejected by EC++ in the context of performance critical, resource constrained, or safety critical tasks. It was therefore decided to write a technical report on "performance" [ISO, 2004]. Its "executive summary reads:

"The aim of this report is:

- to give the reader a model of time and space overheads implied by use of various C++ language and library features,
- to debunk widespread myths about performance problems,

- to present techniques for use of C++ in applications where performance matters, and
- to present techniques for implementing C++ Standard language and library facilities to yield efficient code.

As far as run-time and space performance is concerned, if you can afford to use C for an application, you can afford to use C++ in a style that uses C++'s facilities appropriately for that application."

I strongly recommend this report to people who care about performance, embedded systems, etc. You can download it from http://www.research.att.com/~bs/C++.html.

Not every feature of C++ is efficient and predictable in the sense that we need it for some high performance and embedded applications. In the context of embedded systems, we must consider if we can use

- free store (**new** and **delete**)
- run-time type identification (**dynamic_cast** and **typeid**)
- exceptions (**throw** and **catch**)

Implementations aimed at embedded or high performance applications have compiler options for disabling run-time type identification and exceptions. Free store usage is easily avoided. All other C++ language features are predictable and can be implemented optimally (according to the Zero-overhead principle; see §D&E4.5). Even exceptions tend to be far more efficient than they are reputed to be in some places and should be considered for all but the most stringent hard-real time systems. The TR discusses these issues and defines an interface to the lowest accessible levels of hardware (such as registers).

The performance TR was written by a working group primarily consisting people who cared about embedded systems, including members of the EC++ technical committee. I was active in the performance working group and drafted significant portions of the TR, but the chairman and editor was first Martin O'Riordan and later Lois Goldthwaite. The acknowledgements list 28 people. In 2004, that TR was approved by unanimous vote.

### 3.4.2 The Library TR

When we finished the standard in 1997, we were fully aware that the set of standard libraries was simply the set that we had considered the most urgently needed and also ready to ship. Several much-wanted libraries, such as hash tables, regular expression matching, directory manipulation, and threads, were missing. Work on such libraries started immediately in the Libraries Working Group chaired by Matt Austern (originally working at SGI with Alex Stepanov, then at AT&T Labs with me, and currently at Apple). In 2001, work on a technical report on libraries, providing what people considered the most urgently needed and best specified libraries, was initiated and in 2004 that TR [ISO, 2005] was approved by unanimous vote.

Despite the immense importance of the standard library and its extensions, I will only briefly list the new libraries here:

- Polymorphic Function Object Wrapper
- Tuple Types
- Mathematical Special Functions
- Type Traits
- Regular Expressions
- Enhanced Member Pointer Adaptor
- General Purpose Smart Pointers
- Extensible Random Number Facility
- Reference Wrapper
- Uniform Method for Computing Function Object Return Types
- Enhanced Binder
- Hash Tables

Many of these new library facilities – which can be expected to ship with every new C++ implementation in the future – are obviously "technical"; that is, they exists primarily to support library builders. In particular, they exist to support builders of standard library facilities in the tradition of the STL. The Library TR is not yet officially published so it may take a bit of looking around on the committee's official website (http://www.open-std.org/jtc1/sc22/wg21/) to find the (extensive) details. Here, I will just emphasize three libraries that are of direct interest to large numbers of application builders:

- Regular Expressions
- General Purpose Smart Pointers
- Hash Tables

Regular expression matching is one of the backbones of scripting languages and of much text processing. Finally, C++ has a standard library for that. The central class is **regex (**or rather **basic_regex** matching **basic_string)** providing regular expression matching of patterns compatible with ECMAscript and (with suitable options set) compatible with other popular notations.

The main "Smart pointer" is a reference counted pointer, **shared_ptr,** intended for code where shared ownership is needed. When the last **shared_ptr** to an object is destroyed, the object pointed to is deleted. Smart pointers are popular, but should be approached with care. They are not the panacea that they are sometimes presented to be. In particular, they are far more expensive to use than ordinary pointers, destructors for objects "owned" by a set of **shared_ptr**s will run at unpredictable times, and if a lot of objects are deleted at once because the last **shared_ptr** to them is deleted you can incur "garbage collection delays" exactly as if you were running a general collector. The costs primarily relate to free store allocation of use count objects and especially to locking during access to the use counts in threaded systems. Do not simply replace all your ordinary pointers with **smart_ptr**s if you are concerned with performance or predictability. These concerns kept **smart_ptr**s "ancestor", **counted_ptr**, out of the 1998 standard. If it is garbage collection you want, you might be better off simply using one of the available garbage collectors (http://www.research.att.com/~bs/C++.html).

No such worries affected hash tables; they would have been in the 1998 standard had we had the time to do a proper detailed design and specification job. There were no doubt that a "hash_map" was needed as an optimization to the STL **map** where the key was a character string and we could design a good hash function. The committee didn't have the time, though, and as consequently the Library TR's **unordered_map** (and **unordered_set**) are the result of about 8 years of experiment and industrial use. The name **unordered_map** was chosen because now there are half a dozen incompatible **hash_map**s in use. The **unordered_map** is the result of a consensus among the **hash_map** implementers and their key users in the committee.

The most common reaction to these extensions among developers is "that was about time; why did it take you so long?" and "I want much more right now". That's understandable (I too want much more right now – I just know that I can't get it), but such statements reflects a lack of understanding what an ISO committee is and can do. The committee is run by volunteers and requires both a consensus and an unusual degree of precision of our specifications (see §D&E6.2). The committee doesn't have the millions of dollars that commercial vendors can spend on "free", "standard" libraries for their customers.

## 3.5 What else was going on?

Obviously, most of what goes on in the C++ community happens outside the ISO standards committee. The standards committee is just the focus where changes to the language and the standard library happen. The committee reflects – as it must – trends in the whole software development community. Unfortunately, I cannot survey the trends and events that influenced C++ during the 1995-2004 decade; that would require a whole book. Instead, I'll very briefly mention a handful of key issues.

Java burst onto the programming scene with an unprecedented amount of hype and also an unprecedented amount of marketing aimed at non-programmers. According to some key Sun people (such as Bill Joy), Java was an improved and simplified C++. "What Bjarne would have designed if he hadn't had to be compatible with C" was – and amazingly still is – a frequently heard statement. Java is not that; for example, see §D&E9.2.2. In the light of Java, that section seems more relevant today than when I wrote it.

Unfortunately, the Java proponents and their marketing machines did not limit themselves to hyping the virtues of Java, but stooped to bogus comparisons (e.g., [Gosling,1996]) and name calling of languages seen as competitors (most notably C++). I see Java as a Sun weapon aimed at Microsoft that missed and hit an innocent bystander:

The C++ community. It hurt many smaller language communities even more; consider Smalltalk, Lisp, Eiffel, etc.

Despite many promises, Java didn't replace C++ ("Java will completely kill C++ within two years" was a graphic expression I repeatedly heard in 1996). In fact, the C++ community has trebled in size since the first appearance of Java. Java did, however, do harm to the C++ community by diverting energy and funding away from much needed tools, library and techniques work. Another problem was that Java encouraged a limited "pure object-oriented" view of programming with a heavy emphasis on run-time resolution. This led many C++ programmers to write unnecessarily inelegant and poorly performing code in imitation.

As I predicted, Java has been accreting new features over the years so as to negate its "original virtue" of simplicity, but without catching up on performance. New languages are always claimed to be "simple" and to survive they increase in size and complexity to become useful in real-world applications. Neither Java nor C++ was immune to that effect; the main difference is that I never claimed perfection (or near perfection) for C++. Obviously Java has made great strides in performance – given its initial slowness it couldn't fail to – but so have C++ – and the Java object model inhibits performance where abstraction is seriously used. Basically, C++ and Java are far more different in aims, language structure, and implementation model than most people seem to think.

During the 1995-2004, C also evolved. Unfortunately, C99 [ISO,1999] is in significant ways less compatible with C++ than C89 [ISO,1989] and harder to coexist with. For example, see [Stroustrup, 2002] for a detailed discussion of the C/C++ relationship.

Most C++ libraries are not part of the standard. In fact, there are thousands of C++ libraries "out there" and the lack of coordination within the C++ community is hurting users because these libraries can be hard to find and are rarely built to interoperate. Beman Dawes started an organization called "Boost" (http://www.boost.org) to address part of that problem. Boost is now an active on-line community and a large collection of open-source libraries aimed at augmenting the C++ standard library. Many of the ideas for additions to the standard library are being "field tested" through Boost.

Obviously, I could write thick books about what was and is going on in the C++ community. However, documenting the use of C++ is not the aim of this book. My aim is to present what had direct influence on the language definition. I even interpret "language definition" narrowly to exclude the standard library, except where (as in the case of **string**, **complex**, **iostream**, and the STL) the needs of library design have been a driving force for language design.

In addition to being influenced by events in the software development community, C++ was (and remains) a major influence on the design on new languages, libraries, and tool aimed at practical work. It would be nice if someone would document these influences, but it won't be me.

# 4 Where we might be going – 2005-2014

After year of deliberate inactivity to allow compiler, tools, and library implementers to catch up and for users to absorb the programming techniques supported by Standard C++, the committee is now again considering language extensions. The "extensions working group" has been reconstituted as the "evolution working group". The name change (suggested by Tom Plum) reflects a greater emphasis on the integration of language features and standard library facilities. As ever, I'm the chairman of that working group. We hope that will help ensure a continuity of vision for C++ and a coherence of the final result. Similarly, the committee membership shows a continuous participation of large number of people and organizations. Fortunately, there are also many new faces bringing new interests and new expertise to the committee.

We aim to be cautious and conservative about changes to the language itself, and strongly emphasize compatibility. The aim is to channel the major effort into an expansion of the standard library. In the standard library, we aim to be aggressive and opportunistic.

For the standard library, I hope to build on the momentum from the library TR (§3.3.2) to make it a much broader platform for systems programming. For example, I expect to see libraries for areas such as directory/folder manipulation, threads, and sockets. I also hope that the committee will take pity on the many new C++ programmers and provide library facilities to support novices from a variety of backgrounds (not just beginning programmers and refugees from C). For example, I'd like to see a standard way to use a range checking STL. I have low expectations for the most frequently requested addition to the standard library: a standard GUI (Graphical User Interface). However, miracles sometimes do happen – remember the STL.

For the language itself, I expect to see an emphasis on features that support generic programming because generic programming is the area where our use of the language has progressed the furthest relative to the support offered by the language. Here, I will examine two key areas:

- Concepts: a type system for template arguments
- Initializer lists: a generalization of initialization facilities

As ever, there are far more proposals than the committee could handle or the language could absorb, see for example http://www.research.att.com/~bs/evol-issues.html. Please remember that even accepting just all the good proposals is infeasible.

The overall aim of the language extensions supporting generic programming is to provide greater uniformity of facilities so as to enable the expression of a larger class of problems directly in a generic form.

My other priority (together with better support for generic programming) is better support for beginners. There is a remarkable tendency for proposals to favor the expert users that propose and evaluate them. Something simple that helps only novices for a few months until they become experts is often ignored. I think that's a potentially fatal design bias. Unless novices are sufficiently supported, only few will become experts. Further, many don't want to become experts; they are and want to remain "occasional C++ users". For example, a physicist using C++ for physics calculations or the control of experimental equipment is usually quite happy being a physicist and has only limited time to spend learning programming techniques. As computer scientists we might wish for people to spend more time on programming techniques, but rather than just hoping, we should work on removing unnecessary barriers to adoption of good techniques.

A very simple example is

```
vector<vector<double>> v;
```

In C++98, this is a syntax error because >> is a single lexical token, rather than two >s each closing a template argument list. A correct declaration of **v** would be:

```
vector< vector<double> > v;
```

I consider this an embarrassment. There are perfectly good reasons for the current rule and the evolution working group twice rejected my suggestions that this was a problem that was worth solving. However, those reasons are language technical and of no interest to novices (of all backgrounds – including experts in other languages). Not accepting the first (and most) obvious declaration of **v** wastes time for users and teachers. I expect the >> problem, and many similar "embarrassments" to be absent from C++0x. In fact, together with Francis Glassborow and others, I am trying to systematically eliminate the most frequently occurring such "embarrassments".

Another "embarrassment" is that it is legal to copy an object of a class with a user-defined destructor using a default copy operation (constructor or assignment). Requiring user-defined copy operations in that case would eliminate a lot of nasty errors related to resource management. For example, consider an oversimplified sting class:

```
class String {
public:
	String(char* pp) :sz(strlen(pp)), p(new char[sz+1]) { strcpy(p,pp); }
	~String() { delete[] p; }
	char& operator[](int i) { return p[i]; }
private:
	int sz;
	char* p;
};

void f(char* x)
{
	String s1(x);
```

```
		String s2 = s1;
	}
```

After the construction of **s2**, **s1.p** and **s2.p** points to the same memory, this will be deleted twice, probably with disastrous results. This problem is obvious to the experienced C++ programmer, who will provide proper copy operations or prohibit copying. However, the problem can seriously baffle a novice and undermine trust in the language.

It would be even nicer to ban default copy of objects of a class with pointer members, but that would bring up nasty compatibility problems. Remedying long-standing problems is harder than it looks, especially if C compatibility enters into the picture.

## 4.1 Concepts

The D&E discussion of templates contains three whole pages (§15.4) on constraints on template arguments. Clearly, I felt the need for a better solution. The error messages that come from slight errors in the use of a template, such as a standard library algorithm, can be spectacularly long and unhelpful. The problem is that the template code's expectations of its template arguments are implicit. Consider again **find_if()**:

```
	template<class In, class Pred>
	In find_if(In first, In last, Pred pred)
	{
		while (first!=last && !pred(*first)) ++first;
		return first;
	}
```

Here, we are making a lot of assumptions about the **In** and **Predicate** types. From the code, we can see that **In** must somehow support **!=**, *****, and **++** with suitable semantics and that we must be able to copy **In** objects as arguments and return values. Similarly, we can see that we can call a **Pred** with and argument of whichever type ***** returns from an **In** and apply **!** to the result to get something that can be treated as a Boolean. However, that's all implicit in the code. The standard library carefully documents these requirements for forward iterators (our **In**) and predicates (our **Pred**), but compilers don't read manuals. Try this error and see what your compiler says:

```
	find_if(1,5,3.14);	// errors
```

Partial, but quite effective, solutions based on my old idea of letting a constructor check assumptions about template arguments (§D&E15.4.2) are now finding widespread use. For example:

```
	template<class T> struct Forward_iterator {
		static void constraints(T a) {
			++a; a++;		// can increment
			T b = a; b = a;		// can copy
```

```
                *b = *a;            // can dereference and copy the result
        }
        Forward_iterator() { void (*p)(T) = constraints; }
};
```

This defines a class that will compile only if **T** is a forward iterator. However, a **Forward_iterator** object doesn't really do anything so that compiler can (and does) trivially optimize away such objects. We can use **Forward_iterator** in a definition like this:

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
        Forward_iterator<In>();     // check template argument type
        while (first!=last && !pred(*first)) ++first;
        return first;
}
```

Alex Stepanov and Jeremy Siek did a lot to develop and popularize such techniques. One place where they are used prominently is in the Boost library, but these days you find constraints classes in most standard library implementation. The difference in the quality of error messages is spectacular.

However, constraints classes are at best a partial solution. For example, the testing is done in the definition – it would be much better if the checking could be done given only a declaration. That way, we would obey the usual rules for interfaces and could start considering the possibility of genuine separate compilation of templates.

So, let's tell the compiler what we expect from a template argument:

```
template<Forward_iterator In, Predicate Pred>
In find_if(In first, In last, Pred pred);
```

Assuming that we can express what a **Forward_iterator** and a **Predicate** is, the compiler can now check a call of **find_if()** in isolation from its definition. What we are doing here is to build a type system for template arguments. In the context of modern C++, such "types of types" are called "concepts". There are various ways of specifying such concepts; for now, think of them as a kind of constraints classes with direct language support and a nicer syntax. A concept says what facilities a type must provide, but nothing about how it does provide those facilities. The ideal concept (e.g. **<Forward_iterator In>**) is very close to a mathematical abstraction ("for all types **In** such that an **In** can be incremented, dereferenced, and copied") just as the original **<class T>** is the mathematical "for all types **T**".

Given that only that declaration (and not the definition) of **find_if()**, we can write

```
int x = find_if(1,2,Less_than<int>(7));
```

This call will fail because **int** doesn't support *****. In other words, the call will fail to compile because **int** isn't a **Forward_iterator**. Importantly, that makes it easy for a compiler to report the error in the language of the user and at the point in the compilation where the call is first seen.

Unfortunately, knowing that the iterator arguments are **Forward_iterator**s and that the predicate argument is a **Predicate** isn't enough to guarantee successful compilation of a call of **find_if()**. The two argument types interact. In particular, the predicate takes an argument that is an iterator dereferenced by ***** (**pred(*first)**). Our aim is complete checking of a template in isolation from the calls and complete checking of each call without looking at the template definition, so the concept must be made sufficiently expressive to deal with such interactions among template arguments. One way is to parameterize the concepts in parallel to the way the templates themselves are parameterized. For example:

```
template<Value_type T,
        Forward_iterator<T> In,          // iterates over a sequence of Ts
        Predicate<bool,T> Pred>          // takes a T and returns a bool
In find_if(In first, In last, Pred pred);
```

Here, we require that the **Forward_iterator** must point to elements of a type **T,** which is the same type as the **Predicate**'s argument type.

Expressing required relations among template arguments through common parameters (here, the parameter **T**), is unfortunately not completely expressive, leads to added template parameters, and expresses requirements indirectly. For example, the example above doesn't say that it must be possible to pass the result of ***first** as the argument to **pred**. Instead, it says that **Forward_iterator** and **Predicate** shares a template argument type. To cope with such concerns, we are exploring the possibility of expressing relations among template arguments directly. For example:

```
template<Forward_iterator In, Predicate Pred>
        where (assignable<In::value_type, Pred::argument_type>)
In find_if(In first, In last, Pred pred);
```

This approach has its own problems, such as a tendency of the requirements (the **where** clause) to approximate the template definition itself in complexity and that popular iterators (such as **int***) don't have member types (such as **value_type**).

One possible expression of the idea of a concept is direct support for the kind of expression we are used to with constraints classes. For example, we could define **Forward_iterator** as used in the examples above like this:

```
template <class T> concept Forward_iterator {    // parameterized concept
```

```
        Forward_iterator a;
        ++a; a++;                          // can increment
        Forward_iterator b = a; b = a;     // can copy
        *b = *a;                           // can dereference and copy the result
        T x = *a; *a = x;                  // the result can be treated as a T
    };
```

Or

```
    concept Forward_iterator {         // concept not using parameterization
        Forward_iterator a;
        ++a; a++;                          // can increment
        Forward_iterator b = a; b = a;     // can copy
        *b = *a;                           // can dereference and copy the result
    };
```

The parameterized concept definition would be used with the first declaration of **find_if** and the one without a parameter with the second. They represent alternative language designs. We still have to make design choices in this area. However, consider:

```
    int x = find_if(1,2,Less_than<int>(7));
```

This would be rejected because **1** and **2** are **int**s and **int** doesn't support *. If we used the parameterized concept design, it would also be rejected because **int** isn't a parameterized type matching **Forward_iterator<T>**. On the other hand, consider:

```
    void f(vector<int>& v, int* p, int n)
    {
        vector<int>::iterator q = find_if(v.begin(),v.end(),Less_than<int>(7));
        int* q2 =  find_if(p,p+n,Less_than<int>(7));
        // …
    }
```

This would compile because both **vector<int>::iterator** and **int*** provide all the operations required by the concepts. However, if we used the parameterized concept design, we would need a language rule to allow the compiler to consider **int*** as a **Forward_iterator<T>** with **int** as the argument.

Clearly, I'm reporting work in progress here, but the likelihood is that a form of concepts will be the cornerstone of C++0x. Templates have become essential for the most effective (and efficient) C++ programming styles, but suffer from spectacularly poor error messages, lack of facilities for overloading templates based on template arguments, and poor separate compilation. Concepts directly address all of those concerns without the main weakness of abstract-base class based approaches – the overhead of run-time resolution through virtual function calls (§D&E3.5). Importantly, concepts do not rely on explicitly declared subtype hierarchies, thus not requiring logically redundant hierarchical relationships and allowing built-in types to be considered on equal footing with classes.

There is now an extensive literature on the subject of concepts and their possible relationship to similar constructs in other languages [Stroustrup, 2003a] [Stroustrup, 2003b] [Garcia, 2003]. Matt Austern, Jaako Järvi, Mich Marcus, Gabriel Dos Reis, Jeremy Siek, Alex Stepanov, and I are among the people active with this design problem.

## 4.2 General initializers

One of the fundamental ideas of C++ is to "provide as good support for user-defined types as for built-in types" (§D&E4.4). But consider:

```
double vd[ ] = { 1.2, 2.3, 3.4, 4.5, 5.6 };
vector<double> v(vd, vd+5);
```

We can directly initialize the array with the initializer list, whereas the least bad we can do for a **vector** is to create an array and initialize the **vector** from that array. If there are only few initializer values, I might even prefer to use **push_back()** to avoid explicitly stating the number of initializer values (**5** in the example above):

```
vector<double> v;
v.push_back(1.2);
v.push_back(2.3);
v.push_back(3.4);
v.push_back(4.5);
v.push_back(5.6);
```

I don't think anyone would call either solution pretty. To get the most maintainable code and not favor the built-in (and inherently dangerous) arrays over the recommended user-defined type **vector**, we need to be able to write:

```
vector<double> v = { 1.2, 2.3, 3.4, 4.5, 5.6 };
```

or

```
vector<double> v ({ 1.2, 2.3, 3.4, 4.5, 5.6 });
```

Since argument passing is defined in terms of initialization, this would of course also work for functions taking **vector**s:

```
void f(const vector<double>& r);
// …
f({ 1.2, 2.3, 3.4, 4.5, 5.6 });
```

I believe that this generalization of the use of initializers will be part of C++0x. This would be part of a general overhaul of constructors because people have discovered a number of weaknesses that seems to be amenable to solution through a generalization of constructors, such as forwarding constructors, guaranteed compile-time constructors, and inherited constructors.

## 5 Acknowledgements

Most of my thanks go to the members of the C++ standards committee who made this "extended foreword" necessary. Also thanks to Takanori Adachi, Matt Austern, Andrew Koenig, Gabriel Dos Reis, and Alex Stepanov for constructive comments on an earlier draft of this chapter.

## 6 References

[Austern, 1998] M. Austern: "Generic Programming and the STL: Using and Extending the C++ Standard Template Library". Addison. 1998). ISBN: 0201309564.
[Cargill, 1994] T. Cargill: "Exeption handling: A False Sense of Security". The C++ Report, Volume 6, Number 9, November-December 1994.
[EC++ 1999] the Embedded C++ Technical Committee: "The Language Specification & Libraries Version". WP-AM-003. Oct 1999 (http://www.caravan.net/ec2plus/).
[Garcia, 2003] R. Garcia, et al: "A comparative study of language support for generic programming". ACM OOPSLA 2003.
[Gosling, 1996] Gosling & McGilton: "The Java(tm) Language Environment: A White Paper". http://java.sun.com/docs/white/langenv/
[ISO, 1990] "Standard for the C Programming Language". ISO/IEC 9899. ("C89").
[ISO, 1998] "Standard for the C++ Programming Language". ISO/IEC 14882.
[ISO, 1999] "Standard for the C Programming Language". ISO/IEC 9899:1999. ("C99").
[ISO, 2003a] "The C Standard" (ISO/IEC 9899:2002). Wiley 2003. ISBN 0-470-84573-2.
[ISO, 2003b] "The C++ Standard" (ISO/IEC 14882:2002). Wiley 2003. ISBN 0-470-84674-7.
[ISO, 2004] "Technical Report on C++ Performance" ISO.IEC PDTR 18015
[ISO, 2005] "Technical Report on C++ Standard Library Extensions" ISO/IEC PDTR 19768.
[Stepanov, 1994] A. Stepanov and M. Lee: "The Standard Template Library" HP Labs TR HPL-94-34. August 1994.
[Stepanov, 2004] Alex Stepanov: personal communications.
[Stroustrup,1998] B. Stroustrup: "The C++ Programming Language (3rd Edition)". Addison-Wesley Longman. Reading Mass. USA. 1997. ISBN 0-201-88954-4.
[Stroustrup, 1999] B. Stroustrup: "Learning Standard C++ as a New Language". C/C++ Users Journal. May 1999.
[Stroustrup, 2000] B. Stroustrup: "The C++ Programming Language (Special Edition)". Addison Wesley. Reading Mass. USA. February 2000. ISBN 0-201-70073-5.
[Stroustrup, 2002] B. Stroustrup: "C and C++: Siblings", "C and C++: A Case for Compatibility", "C and C++: Case Studies in Compatibility". The C/C++ Users Journal. July, August, and September 2002.
[Stroustrup,2003a] B. Stroustrup: "Concept checking - A more abstract complement to type checking". C++ standard committee. Paper N1510.

[Stroustrup,2003b] B. Stroustrup, G. Dos Reis: “Concepts - Design choices for template argument checking” C++ standard committee. Paper N1522.