

Direction for C++0x

Bjarne Stroustrup
Texas A&M University
(and AT&T – Research)
<http://www.research.att.com>

Abstract

A good programming language is far more than a simple collection of features. My ideal is to provide a set of facilities that smoothly work together to support design and programming styles of a generality beyond my imagination. Here, I outline rules of thumb (guidelines, principles) that are being applied in the design of C++0x. For example, generality is preferred over specialization, novices as well as experts are supported, library extensions are preferred over language changes, compatibility with C++98 is emphasized, and evolution is preferred over radical breaks with the past. Since principles cannot be understood in isolation, I very briefly present a few of the proposals such as concepts, generalized initialization, auto, template aliases, being considered in the ISO C++ standards committee.

Overview

- The problem
- Standardization
- Rules of thumb
- Examples
- Summaries

ISO Standard C++

- C++ is a general-purpose programming language with a bias towards systems programming that
 - is a better C
 - supports data abstraction
 - supports object-oriented programming
 - supports generic programming
- A multi-paradigm programming language
 - (if you must use long words)
 - The most effective styles use a combination of techniques

Problems

- C++ is immensely popular
 - well over 3 million programmers according to IDC
 - incredibly diverse user population
 - Application areas
 - Programmer ability
- Many people want improvements (of course)
 - For people like them doing work like them
 - “just like language XYZ”
 - And don’t increase the size of the language, it’s too big already
- Many people absolutely need stability
 - N*100M lines of code

Problems

- We can’t please everyone
 - The list of requested features in large and growing
 - See my C++ page
 - The language really is uncomfortably large and complex
- A language is far more than a simple collection of features
 - Designing a language feature to fit into a language is hard
 - Generality
 - Composability
 - Adding a feature can harm users
 - Performance
 - compile time, run time
 - Compatibility
 - Source, linkage, ABIs
 - Ease of learning

The (real) problems

- Help people to write better programs
 - Easier to write
 - Easier to maintain
 - Easier to achieve acceptable resource usage

C++ ISO Standardization

- Current status
 - ISO standard 1998, TC 2003,
 - Library TR 2005, Performance TR 2005
 - C++0x in the works – due 200x
- Membership
 - About 22 nations (8 to 12 represented at each meeting)
 - ANSI hosts the technical meetings
 - Other nations have further technical meetings
 - About 120 active members (50+ at each meeting)
 - About 200 members in all
 - Down ~40% from its height (1996), up again the last few years
- Process
 - formal, slow, bureaucratic, and democratic
 - “the worst way, except for all the rest” (apologies to W. Churchill)

Standardization – why bother?

- Directly affects millions
 - Huge potential for improvement
 - So much code is appallingly poor
- Defense against vendor lock-in
 - Only a partial defense, of course
 - I really don't like proprietary languages
- There are still many new techniques to get into use
 - They require language or standard library support to affect mainstream use
- For C++, the ISO standards process is central
 - C++ has no rich owner who dictates changes or controls a tame standards progress
 - And pays for marketing
 - The C++ standards committee is the central forum of the C++ community
 - For (too) many: “if it isn't in the standard it doesn't exist”
 - Unfair, but a reality

Why mess with a good thing?

- The ISO Standard is good
 - but not perfect
- ISO rules require review
 - Community demands consideration of new ideas
- We face increasingly difficult tasks
 - We == programmers and system designers
- The world changes
 - and poses new challenges
- We have learned a lot since 1996
 - When the last of the ISO C++ features was proposed
- Stability is good
 - but the computing world craves novelty
 - Without challenges, the best people will depart for greener pastures

Overall Goals

- Make C++ a better language for systems programming and library building
 - Rather than providing specialized facilities for a particular sub-community (e.g. numeric computation or Windows-style application development)
- Make C++ easier to teach and learn
 - Through increased uniformity, stronger guarantees, and facilities supportive of novices (there will always be more novices than experts)

Rules of thumb / Ideals

- Provide stability and compatibility
- Prefer libraries to language extensions
- Make only changes that changes the way people think
- Prefer generality to specialization
- Support both experts and novices
- Increase type safety
- Improve performance and ability to work directly with hardware
- Fit into the real world

Stability and compatibility

- The aim for C++0x is evolution constrained by a strong need for compatibility.
- The aim of that evolution is to provide major real-world improvements.
 - Not fiddling with minor details
- 100% compatibility is too constraining
 - E.g. new keyword
 - `static_assert`
 - We avoid extreme circumlocution
 - `#define static_assert __Static_assert`

Libraries and language features

- Prefer libraries to language extensions
- A major aim of the language is to support better library building
 - Well-defined machine model
 - Better support for generic programming
 - Move semantics
- New library component examples
 - `Unordered_map` (`hash_map`; Library TR 2004)
 - `Regexp` (Library TR – 2004)
 - “smart” pointers (Library TR – 2004)
 - File manipulation
 - Threads

Prefer generality to specialization

- The aim for C++0x is to supply general language mechanisms that can be used freely in combination and to deliver more specialized features as standard library facilities built from language features available to all.
- Examples
 - Better generic programming support
 - Improve initialization facilities
 - Provide user-defined constant expressions (ROMable)
- C++ will remain a general-purpose language
 - Not, a specialized
 - web language,
 - a Windows application language
 - embedded systems programming language
 - We'll be better in all of those application areas – and more

Support novices

- C++ has become too “expert friendly”
- Most of us are novices at something most of the time
- Have you ever written something like this?

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

or this?

```
int i = extract_int(s);    // s is a string, e.g. “12.37”
```

or this?

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


Better (C++0x)

- This'll work
vector<vector<double>> v; // no space between the >s

auto p = find(tbl.begin(), tbl.end(), x);
// tbl is a const vector<int>
// p becomes vector<int>::const_iterator
- The >> and **auto** solutions have been approved for C++0x
- “Supporting novices of all backgrounds” requires work on both the language and the standard library.
- Concerns for education will be central for that
 - E.g., “Learning Standard C++ as a new Language” [Stroustrup, 1999].
- Overloading based on concepts, will allow a further simplification

auto p = find(tbl, x); // tbl is some container

Type safety

- For correctness, safety and security, and convenience
 - complex, dangerous code:


```
void get_input(char* p)
{
    char ch;
    while (cin.get(ch) && !iswhite(ch)) *p++ = ch;
    *p = 0;
}
```
 - Better, much better:


```
string s;
cin >> s;
```

Type safety

- For performance
 - Messy, slow code:

```
struct Link {  
    Link* link;  
    void* data;  
};  
  
void my_clear(Link* p, int sz)    // clear data of size sz  
{  
    for (Link* q = p; q!=0; q = q->link) memset(q->data,0,sz);  
}
```

- Simpler, faster code:

```
template<class In> void my_stl_clear(In first, In last)  
{  
    while (first!=last) *first++ = 0;  
}
```

Areas of language change

- Machine model and concurrency
- Modules and libraries
- Concepts and other type stuff
 - Auto, decltype, template aliases, “strong enums”
 - initialization
- Etc.
 - >>, static_assert, long long, for each, C99 character types

C++98 example

- Initialize a vector
 - clumsy

```
template<class T> class vector {  
    // ...  
    void push_back(const T&) { /* ... */ }  
    // ...  
};  
  
vector<double> v;  
v.push_back(1.2);  
v.push_back(2.3);  
v.push_back(3.4);
```

C++98 example

- Initialize a vector
 - Awkward
 - Spurious use of (unsafe) array

```
template<class T> class vector {  
    // ...  
    template <class Input_iterator Iter>  
        void vector(Iter first, Iter last) { /* ... */ }  
    // ...  
};  
  
int a[ ] = { 1.2, 2.3, 3.4 };  
vector<double> v(a, a+sizeof(a)/sizeof(int));
```

- Important principle (currently violated):
 - Support user-defined types as well as built-in types

C++0x version

```
template<Value_type T> class vector {    // note: T is typed
    // ...
    vector(Sequence<T>);    // sequence constructor
    // ...
};
vector<double> v = { 1.2, 2.3, 3.4 };
```

- Exactly how should the sequence constructor be defined?

Will this happen?

- Probably
 - Lillehammer meeting adopted schedule aimed at ratified standard in 2009 (feature complete late 2007)
 - With the feature set as described here
 - We'll be flooded with new request before August 2005 "proposal freeze"
 - We'll slip up a few times – this really is hard
 - Ambitious, but
 - We'll work harder
 - We have done it before

Core language suggestions (Lots!)

- **decltype/auto** – type deduction from expressions
- Template alias
- **#nomacro**
- Extern template
- Dynamic library support
- Allow local classes as template parameters
- Move semantics
- **nullptr** - Null pointer constant
- Static assertions
- Concepts (a type system for types)
- Solve the forwarding problem
- Variable-length template parameter lists
- Simple compile-time reflection
- GUI support (e.g. slots and signals)
- Defaulting and inhibiting common operations
- Class namespaces
- **long long**
- >> (without a space) to terminate two template specializations
- ...

Library TR

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

What's out there? (Lots!)

Library building is the most fertile source of ideas

- Libraries
- Core language

- Boost.org – libraries loosely based on the standard libraries
- ACE – portable distributed systems programming platform
- Blitz++ – the original template-expression linear-algebra library
- SI – statically checked international units
- Loki – mixed bag of very clever utility stuff
- Endless GUIs and GUI toolkits
 - GTK+/gtkmm, Qt, FOX Toolkit, eclipse, FLTK, wxWindows, ...
- ... much, much more ...

see the C++ libraries FAQ

- Link on <http://www.research.att.com/~bs/C++.html>

What's out there? Boost.org

- Filesystem Library – Portable paths, iteration over directories, etc
- MPL added – Template metaprogramming framework
- Spirit Library – LL parser framework
- Smart Pointers Library –
- Date-Time Library –
- Function Library – function objects
- Signals – signals & slots callbacks
- Graph library –
- Test Library –
- Regex Library – regular expressions
- Format Library added – Type-safe 'printf-like' format operations
- Multi-array Library added – Multidimensional containers and adaptors
- Python Library – reflects C++ classes and functions into Python
- uBLAS Library added – Basic linear algebra for dense, packed and sparse matrices
- Lambda Library – `for_each(a.begin(), a.end(), std::cout << _1 << ' ');`
- Random Number Library
- Threads Library
- ...

Performance TR

- 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++ language and standard library facilities to yield efficient code.
- Contents
 - Language features: overheads and strategies
 - Creating efficient libraries
 - Using C++ in embedded systems
 - Hardware addressing interface