C++11 - the new ISO C++ standard

Modified August 19, 2016

This document is written by and maintained by Bjarne Stroustrup. Constructive comments, corrections, references, and suggestions are of course most welcome. Currently, I'm working to improve completeness and clean up the references.

Translations:

- Russian.
- Chinese.
- Japanese.
- Korean

I have contributed to the new, unified, isocpp.org C++ FAQ maintained by The C++ Foundation of which I am a director. The maintenance of this FAQ is likely to become increasingly sporatic.

C++11 is the ISO C++ standard ratified in 2011. The previous standard is often referred to as C++98 or C++03; the differences between C++98 and C++03 are so few and so technical that they ought not concern users.

A late working paper is available. This is close to the final draft international standard formally accepted by a 21-0 national vote in August 2011.

Before its official ratification, we called the upcoming standard C++0x. I have not had the time to update the name consistently, sorry, and anyway I like the name C++0x :-). The name "C++0x" is a relict of the days where I and others, hoped for a C++08 or C++09. Think of 'x' as hexadecimal (i.e., C++0B == C++11).

All official documents relating to C++11/C++0x can be found at the ISO C++ committee's website. The official name of the committee is SC22 WG21.

Caveat: This FAQ will be under construction for quite a while. Comments, questions, references, corrections, and suggestions welcome.

Purpose

The purpose of this C++11 FAQ is

- To give an overview of the new facilities (language features and standard libraries) offered by C++11 in addition to what is provided by the previous version of the ISO C++ standard.
- To give an idea of the aims of the ISO C++ standards effort.
- To present a user's view of the new facilities
- To provide references to allow for a more in depth study of features.
- To name many of the individuals who contributed (mostly as authors of the reports they wrote for the committee). The standard is not written by a faceless organization.

Please note that the purpose of this FAQ is not to provide comprehensive discussion of individual features or a detailed explanation of how to use them. The aim is to give simple examples to demonstrate what C++11 has to offer (plus references). My ideal is "max one page per feature" independently of how complex a feature is. Details can often be found in the references.

Lists of questions

Here are some high-level questions

Questions about individual language features can be found here:

- long long integers (at least 64 bits)
- memory model
- move semantics; see rvalue references
- narrowing (how to prevent it)
- **[[noreturn]]**
- null pointer (**nullptr**)
- override controls: **override**
- override controls: **final**
- PODs (generalized)
- range-for statement
- raw string literals
- right-angle brackets
- rvalue references
- Simple SFINAE rule
- static (compile-time) assertions (**static_assert**)
- template alias
- template typedef; see template alias
- thread-local storage (**thread_local**)
- unicode characters
- Uniform initialization syntax and semantics
- unions (generalized)
- user-defined literals
- variadic templates

I often borrow examples from the proposals. In those cases: Thanks to the proposal authors. Many of the examples are borrowed from my own talks and papers.

Questions about individual standard library facilities can be found here:

- abandoning a process
- Improvements to algorithms
- array
- async()
- atomic operations
- Condition variables
- Improvements to containers
- function and bind
- forward_list a singly-liked list
- future and promise
- garbage collection ABI
- hash_tables; see unordered_map
- metaprogramming and type traits
- Mutual exclusion
- random number generators
- regex a regular expression library
- scoped allocators
- shared_ptr
- smart pointers; see shared_ptr, weak_ptr, and unique_ptr
- threads
- Time utilities
- tuple
- unique_ptr
- unordered_map
- weak_ptr
- system error

Below are answers to specific questions as indexed above.

What do *you* think of C++11?

That's a (to me) amazingly frequent question. It may be the most frequently asked question. Surprisingly, C++11 feels like a new language: The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever. If you timidly approach C++ as just a better C or as an object-oriented language, you are going to miss the point. The abstractions are simply more flexible and affordable than before. Rely on the old mantra: If you think of *it* as a separate idea or object, represent it directly in the program; model real-world objects, and abstractions directly in code. It's easier now: Your ideas will map to enumerations, objects, classes (e.g. control of defaults), class hierarchies (e.g. inherited constructors), templates, aliases, exceptions, loops, threads, etc., rather than to a single "one size fits all" abstraction mechanism.

My ideal is to use programming language facilities to help programmers think differently about system design and implementation. I think C++11 can do that - and do it not just for C++ programmers but for programmers used to a variety of modern programming languages in the general and very broad area of systems programming.

In other words, I'm still an optimist.

When will C++11 be a formal standard?

It is now! The first draft for formal comments was produced in September 2008. The Final International Draft standard (FCD) unanimously approved by the ISO C++ committee on March 25, 2011. It was formally approved by a 21-0 national vote in August 2011. The standard was published this year (2011).

Following convention, the new standard is called C++11 (because it was published in 2011). Personally, I prefer plain C++ and to use a year marker only when I need to distinguish it from previous versions of C++, such as ARM C++, C++98 and C++03. For a transition period, I still use C++0x in places. Think of 'x' as hexadecimal.

When will compilers implement C++11?

Currently shipping compilers (e.g. GCC C++, Clang C++,IBM C++, and Microsoft C++) already implement many C++11 features. For example, it seems obvious and popular to ship all or most of the new standard libraries.

I expect more and more features to become available with each new release. I expect to see the first complete C++11 compiler sometime in 2012, but I do not care to guess when such a compiler ships or when every compiler will provide all of C++11. I note that every C++11 feature has already been implemented by someone somewhere so there is implementation experience available for implementers to rely on.

Here are links to C++11 information from purveyors:

- comparison
- GCC
- IBM
- Microsoft
- EDG
- Clang

When will the new standard libraries be available?

Initial versions of the new standard libraries are currently shipping with the GCC, Clang and Microsoft implementations, and are available from boost.

What new language features does C++11 provide?

You don't improve a language by simply adding every feature that someone considers a good idea. In fact, essentially every feature of most modern languages has been suggested to me for C++ by someone: Try to imagine what the superset of C99, C#, Java, Haskell, Lisp, Python, and Ada would look like. To make the problem more difficult, remember that it is not feasible to eliminate older features, even if the committee agrees that they are bad: experience shows that users force every implementer to keep providing deprecated and banned features under compatibility switches (or by default) for decades.

To try to select rationally from the flood of suggestions we devised a set of specific design aims. We couldn't completely follow them and they weren't sufficiently complete to guide the committee in every detail (and IMO couldn't possible be that complete).

The result has been a language with greatly improved abstraction mechanisms. The range of abstractions that C++ can express elegantly, flexibly, and at zero costs compared to hand-crafted specialized code has greatly increased. When we say "abstraction" people often just think "classes" or "objects." C++11 goes far beyond that: The range of user-defined types that can be cleanly and safely expressed has grown with the addition of features such as initializer-lists, uniform initialization, template aliases, rvalue references, defaulted and deleted functions, and variadic templates. Their implementation eased with features, such as auto, inherited constructors, and decltype. These enhancements are sufficient to make C++11 feel like a new language.

For a list of accepted language features, see the feature list

What new standard libraries does C++11 provide?

I would have liked to see more standard libraries. However, note that the standard library definition is already about 70% of the normative text of the standard (and that doesn't count the C standard library, which is included by reference). Even though some of us would have liked to see many more standard libraries, nobody could claim that the Library working group has been lazy. It is also worth noting that the C++98 libraries have been significantly improved through the use of new language features, such as initializer-lists, rvalue references, variadic templates, noexcept, and constexpr. The C++11 standard library is easier to use and provides better performance that the C++98 one.

For a list of accepted libraries, see the library component list.

What were the aims of the C++11 effort?

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

The overall aims of the C++11 effort was to strengthen that:

- Make C++ a better language for systems programming and library building -- that is, to build directly on C++'s contributions to programming, 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).

Naturally, this is done under very stringent compatibility constraints. Only very rarely is the committee willing to break standards conforming code, though that's done when a new keyword (e.g. static_assert, nullptr, and constexpr) is introduced.

For more details see:

- B. Stroustrup: What is C++11?. CVu. Vol 21, Issues 4 and 5. 2009.
- B. Stroustrup: Evolving a language in and for the real world: C++ 1991-2006. ACM HOPL-III. June 2007.
- B. Stroustrup: A History of C++: 1979-1991. Proc ACM History of Programming Languages conference (HOPL-2). March 1993.
- B. Stroustrup: C and C++: Siblings. The C/C++ Users Journal. July 2002.

What specific design aims guided the committee?

Naturally, different people and different organizations involved with the standardization have somewhat different aims, especially when it comes to details and to priorities. Also, detailed aims change over time. Please remember that the committee can't even do all that everyone agrees would be good things -- it consists of volunteers with very limited resources. However, here are a set of criteria that has seen real use in the discussion of which features and libraries were appropriate for C++11:

- Maintain stability and compatibility -- don't break old code, and if you absolutely must, don't break it quietly.
- Prefer libraries to language extensions -- an ideal at which the committee wasn't all that successful; too many people in the committee and elsewhere prefer "real language features."
- Prefer generality to specialization -- focus on improving the abstraction mechanisms (classes, templates, etc.).
- Support both experts and novices -- novices can be helped by better libraries and through more general rules; experts need general and efficient features.
- Increase type safety -- primarily though facilities that allow programmers to avoid type-unsafe features.
- Improve performance and ability to work directly with hardware -- make C++ even better for embedded systems programming and high-performance computation.
- Fit into the real world -- consider tool chains, implementation cost, transition problems, ABI issues, teaching and learning, etc.

Note that integrating features (new and old) to work in combination is the key -- and most of the work. The whole is much more than the simple sum of its parts.

Another way of looking at detailed aims is to look at areas of use and styles of usage:

- Machine model and concurrency -- provide stronger guarantees for and better facilities for using modern hardware (e.g. multicores and weakly coherent memory models). Examples are the thread ABI, futures, thread-local storage, and the atomics ABI.
- Generic programming -- GP is among the great success stories of C++98; we needed to improve support for it based on experience. Examples are auto and template aliases.
- Systems programming -- improve the support for close-to-the-hardware programming (e.g. low-level embedded systems programming) and efficiency. Examples are constexpr, std::array, and generalized PODs.
- Library building -- remove limitations, inefficiencies, and irregularities from the abstraction mechanisms. Examples are inline namespace, inherited constructors, and rvalue references.

Where can I find the committee papers?

Go to the papers section of the committee's website. There you will most likely drown in details. Look for "issues lists" and "State of " (e.g. State of Evolution (July 2008)) lists. The key groups are

- Core (CWG) -- dealing with language-technical issues and formulation
- Evolution (EWG) -- dealing with language feature proposals and issues crossing the language/library boundary
- Library (LWG) -- dealing with library facility proposals

Here is the latest draft C++11 standard.

Where can I find academic and technical papers about C++11?

- Bjarne Stroustrup: Software Development for Infrastructure. Computer, vol. 45, no. 1, pp. 47-58, Jan. 2012, doi:10.1109/MC.2011.353. A video interview about that paper and video of a talk on a very similar topic (That's a 90 minute talk incl. Q&A).
- Saeed Amrollahi: Modern Programming in the New Millenium: A Technical Survey on Outstanding features of C++0x. Computer Report (Gozaresh-e Computer),

No.199, November 2011 (Mehr and Aban 1390), pages 60-82. (in Persian)
- Mark Batty et al's: Mathematizing C++ concurrency, POPL 2012. // thorough, precise, and mathematical.
- Gabriel Dos Reis and Bjarne Stroustrup: General Constant Expressions for System Programming Languages. SAC-2010. The 25th ACM Symposium On Applied Computing.
- Hans-J. Boehm and Sarita V. Adve: Foundations of the C++ concurrency memory model. ACM PLDI'08.
- Hans-J. Boehm: Threads Basic. HPL technical report 2009-259 // ``what every programmer should know about memory model issues''
- Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine: Concepts: Linguistic Support for Generic Programming in C++. OOPSLA'06, October 2006. // The concept design and implementation as it stood in 2006; it has improved since, though not sufficiently to save it.
- Douglas Gregor and Jaakko Jarvi: Variadic templates for C++0x. Journal of Object Technology, 7(2):31-51, February 2008.
- Jaakko Jarvi and John Freeman: Lambda functions for C++0x. ACM SAC '08.
- Jaakko Jarvi, Mat Marcus, and Jacob N. Smith: Programming with C++ Concepts. Science of Computer Programming, 2008. To appear.
- M. Paterno and W. E. Brown : Improving Standard C++ for the Physics Community. CHEP'04. // Much have been improved since then!
- Michael Spertus and Hans J. Boehm: The Status of Garbage Collection in C++0X. ACM ISMM'09.
- Verity Stob: An unthinking programmer's guide to the new C++ -- Raising the standard. The Register. May 2009. (Humor (I hope)).
- [N1781=05-0041] Bjarne Stroustrup: Rules of thumb for the design of C++0x.
- Bjarne Stroustrup: Evolving a language in and for the real world: C++ 1991-2006. ACM HOPL-III. June 2007. (incl. slides and videos). // Covers the design aims of C++0x, the standards process, and the progress up until 2007.
- B. Stroustrup: What is C++0x?. CVu. Vol 21, Issues 4 and 5. 2009.
- Anthony Williams: Simpler Multithreading in C++0x. devx.com.

This list is likely to be incomplete -- and likely to frequently go out of date as people write new papers. If you find a paper that ought to be here and is not, please send it. Also, not all papers will be completely up-to-date with the latest improvements of the standard. I'll try to keep comments current.

Where else can I read about C++11?

The amount of information about C++11 is increasing as the standard nears completion and C++ implementations start providing new language features and libraries. Here is a short list of sources:

- B. Stroustrup: The C++ Programming Language (Fourth Edition).
- the papers section of the committee's website.
- C++11 draft.
- the C++11 Wikipedia entry. Seems to be actively maintained, though apparently not by members of the committee.
- A list of support for C++11 features.

Are there any videos about C++11?

(To people who know me, this is a proof that this really is an FAQ, rather than a series of my own favorite questions; I'm not a fan of videos on technical topics -- I find the video distracting and the verbal format too likely to contain minor technical errors).

Yes:

- B. Stroustrup, H. Sutter, H-J. Boehm, A. Alexandrescu, S.T.Lavavej, Chandler Carruth, Andrew Sutter, and more: several talks and panels from the Going Native 2012 conference.
- B. Stroustrup, H. Sutter, Sean Parent, Scott Meyers, and more: several talks and panels from the Going Native 2013 conference.
- Herb Sutter: Writing modern C++ code: how C++ has evolved over the years. September 2011.
- Herb Sutter: C++ and Beyond 2011: Herb Sutter - Why C++?. August 2011.
- Try Google videos.
- Lawrence Crowl: Lawrence Crowl on C++ Threads. in Sophia Antipolis, June 2008.
- Bjarne Stroustrup: The design of C++0x at U of Waterloo in 2007.
- Bjarne Stroustrup: Initialization at Google in 2007.
- Bjarne Stroustrup: C++0x -- An overview. in Sophia Antipolis, June 2008.
- Lawrence Crowl: Threads.
- Roger Orr: C++0x. January 2008.
- Hans-Jurgen Boehm: Getting C++ Threads Right. December 2007.

Is C++11 hard to learn?

Well, since we can't remove any significant features from C++ without breaking large amounts of code, C++11 is larger than C++98, so if you want to know every rule, learning C++11 will be harder. This leaves us with just two tools for simplification (from the point of view of learners):

- Generalization: Replace, say, three rules with one more general rule (e.g., uniform initialization, inheriting constructors, and threads).
- Simpler alternatives: Provide new facilities that are easier to use than their older alternatives (e.g., the array, auto, range-for statement, and regex).

Obviously, a "bottom up" teaching/learning style will nullify any such advantage, and there are currently (obviously) very little material that takes a different approach. That ought to change with time.

How does the committee operate?

The ISO Standards committee, SC22 WG21, operates under the ISO rules for such committees. Curiously enough, these rules are not standardized and change over time.

Many countries have national standards bodies with active C++ groups. These groups hold meetings, coordinate over the web, and some send representatives to the ISO meetings. Canada, France, Germany, Switzerland, UK, and USA are present at most meetings. Denmark, the Netherlands, Japan, Norway, Spain, and others are represented in person less frequently.

Much of the work goes on in-between meetings over the web and the results are recorded as numbered committee papers on the WG21 website.

The committee meets two to three times a year for a week each time. Most work at those meetings are in sub-working groups, such as "Core", "Library", "Evolution", and "Concurrency." As needed, there are also in-between meetings of ad-hoc working groups on specific urgent topics, such as "concepts" and "memory model." Voting takes place at the main meetings. First, working groups hold "straw votes" to see if an issue is ready for presentation to the committee as a whole. Then, the committee as a whole votes (one member one vote) and if something is accepted the nations vote. We take great care that we do not get into a situation where the majority present and the nations disagrees -- proceeding if that is the case would guarantee long-term controversy. Final votes on official drafts are done by mail by the national standards bodies.

The committee has formal liaison with the C standards group (SC22 WG14) and POSIX, and more or less formal contacts with several other groups.

Who is on the committee?

The committee consists of a large number of people (about 250) out of whom 90+ turn up at the week-long meetings two or three times a year. In addition there are national standards groups and meetings in several countries. Most members contribute either by attending meetings, by taking part in email discussions, or by submitting papers for committee consideration. Most members have friends and colleagues who help them. From day #1, the committee has had members from many countries and at every meeting people from half a dozen to a dozen countries attend. The final votes are done by about 20 national standards bodies. Thus, the ISO C++ standardization is a fairly massive effort, *not* a small coherent group of people working to create a perfect language for "people just like themselves." The standard is what this group of volunteers can agree on as being the best they can produce that *all* can live with.

Naturally, many (but not all) of these volunteers have day jobs focused on C++: We have compiler writers, tool builders, library writers, application builders (too few of those), researchers (only a few), consultants, test-suite builders, and more.

Here is a very abbreviated list of organizations involved: Adobe, Apple, Boost, Bloomberg, EDG, Google, HP, IBM, Intel, Microsoft, Red Hat, Sun.

Here is a short list of names of members who you may have encountered in the literature or on the web: Dave Abrahams, Matt Austern, Pete Becker, Hans Boehm, Steve

Clamage, Lawrence Crowl, Beman Dawes, Francis Glassborow, Doug Gregor, Pablo Halpern, Howard Hinnant, Jaakko Jarvi, John Lakos, Alisdair Meredith, Jens Maurer, Jason Merrill, Sean Parent, P.J. Plauger, Tom Plum, Gabriel Dos Reis, Bjarne Stroustrup, Herb Sutter, David Vandevoorde, Michael Wong. Apologies to the 200+ current and past members that I couldn't list. Also, please note the author lists on the various papers: a standard is written by (many) individuals, not by an anonymous committee.

You can get a better impression of the breath and depth of expertise involved by examining the author lists on the WG21 papers, but please remember there are major contributors to the standards effort who do not write a lot.

Will there be a C++1y?

Almost certainly -- and not just because the committee has slipped the deadline for C++0x. The plans for minor revisions, C++14, are well advanced (the features have been voted into the working draft and implemented), and the plan is for a major revision in 2017, C++17.

What happened to "concepts"?

"Concepts" was a feature designed to allow precise specification of requirements on template arguments. Unfortunately, the committee decided that further work on concepts could seriously delay the standard and voted to remove the feature from the working paper, see my note The C++0x "Remove Concepts" Decision and A DevX interview on concepts and the implications for C++0x for an explanation.

A radically simplified version ``concepts lite'' will be part of C++14 (as a technical report).

I have not deleted the concept sections from this document, but left them at the end:

- axioms (semantic assumptions)
- concepts
- concept maps

Are there any features you don't like?

Yes. There are also features in C++98 that I don't like, such as macros. The issue is not whether I like something or if I find it useful for something I want to do. The issue is whether someone has felt enough of a need to convince others to support the idea or possibly if some usage is so ingrained in a user community that it needs support.

__cplusplus

In C++11 the macro **__cplusplus** will be set to a value that differs from (is greater than) the current **199711L**.

---

auto -- deduction of a type from an initializer

Consider

```
auto x = 7;
```

Here **x** will have the type **int** because that's the type of its initializer. In general, we can write

```
auto x = expression;
```

and the type of **x** will be the type of the value computed from "expression".

The use of **auto** to deduce the type of a variable from its initializer is obviously most useful when that type is either hard to know exactly or hard to write. Consider:

```
template<class T> void printall(const vector<T>& v)
{
	for (auto p = v.begin(); p!=v.end(); ++p)
		cout << *p << "\n";
}
```

In C++98, we'd have to write

```
template<class T> void printall(const vector<T>& v)
{
	for (typename vector<T>::const_iterator p = v.begin(); p!=v.end(); ++p)
		cout << *p << "\n";
}
```

When the type of a variable depends critically on template argument it can be really hard to write code without **auto**. For example:

```
template<class T, class U> void multiply(const vector<T>& vt, const vector<U>& vu)
{
	// ...
	auto tmp = vt[i]*vu[i];
	// ...
}
```

The type of **tmp** should be what you get from multiplying a **T** by a **U**, but exactly what that is can be hard for the human reader to figure out, but of course the compiler knows once it has figured out what particular **T** and **U** it is dealing with.

The **auto** feature has the distinction to be the earliest to be suggested and implemented: I had it working in my Cfront implementation in early 1984, but was forced to take it out because of C compatibility problems. Those compatibility problems disappeared when C++98 and C99 accepted the removal of "implicit **int**"; that is, both languages require every variable and function to be defined with an explicit type. The old meaning of **auto** ("this is a local variable") is now illegal. Several committee members trawled through millions of lines of code finding only a handful of uses -- and most of those were in test suites or appeared to be bugs.

Being primarily a facility to simplify notation in code, **auto** does not affect the standard library specification.

See also

- the C++ draft section 7.1.6.2, 7.1.6.4, 8.3.5 (for return types)
- [N1984=06-0054] Jaakko Jarvi, Bjarne Stroustrup, and Gabriel Dos Reis: Deducing the type of variable from its initializer expression (revision 4).

Range-for statement

A range for statement allows you to iterate through a "range", which is anything you can iterate through like an STL-sequence defined by a begin() and end(). All standard containers can be used as a range, as can a std::string, an initializer list, an array, and anything for which you define begin() and end(), e.g. an istream. For example:

```
void f(vector<double>& v)
{
	for (auto x : v) cout << x << '\n';
	for (auto& x : v) ++x;	// using a reference to allow us to change the value
}
```

You can read that as "for all x in v" going through starting with v.begin() and iterating to v.end(). Another example:

```
for (const auto x : { 1,2,3,5,8,13,21,34 }) cout << x << '\n';
```

The **begin()** (and **end()**) can be a member to be called **x.begin()** or a free-standing function to be called **begin(x)**. The member version takes precedence.

See also

- the C++ draft section 6.5.4 (note: changed not to use concepts)
- [N2243==07-0103] Thorsten Ottosen: Wording for range-based for-loop (revision 2).
- [N3257=11-0027] Jonathan Wakely and Bjarne Stroustrup: Range-based for statements and ADL (Option 5 was chosen).

---

right-angle brackets

Consider

```
list<vector<string>> lvs;
```

In C++98 this is a syntax error because there is no space between the two >s. C++11 recognizes such two **>**s as a correct termination of two template argument lists.

Why was this ever a problem? A compiler front-end is organized parses/stages. This is about the simplest model:

- lexical analysis (make up tokens from characters)
- syntax analysis (check the grammar)
- type checking (find the type of names and expressions)

These stages are in theory and sometimes in practice strictly separate, so the lexical analyzer that determines that **>>** is a token (usually meaning right-shift or input) has no idea of its meaning; in particular, it has no idea of templates or nested template argument lists. However, to get that example "correct" the three stages has somehow to cooperate. The key observation that led to the problem being resolved was that every C++ compiler already did understand the problem so that it could give decent error messages.

See also

- the C++ draft section ???
- [N1757==05-0017] Daveed Vandevoorde: revised right angle brackets proposal (revision 2).

---

control of defaults: default and delete

The common idiom of "prohibiting copying" can now be expressed directly:

```
class X {
	// ...
	X& operator=(const X&) = delete;	// Disallow copying
	X(const X&) = delete;
};
```

Conversely, we can also say explicitly that we want to default copy behavior:

```
class Y {
	// ...
	Y& operator=(const Y&) = default;	// default copy semantics
	Y(const Y&) = default;
};
```

Being explicit about the default is redundant. However, comments about copy operations and (worse) a user explicitly defining copy operations meant to give the default behavior are not uncommon. Leaving it to the compiler to implement the default behavior is simpler, less error-prone, and often leads to better object code.

The "default" mechanism can be used for any function that has a default. The "delete" mechanism can be used for any function. For example, we can eliminate an undesired conversion like this:

```
struct Z {
	// ...

	Z(long long);     // can initialize with an long long
	Z(long) = delete; // but not anything less
};
```

See also

- the C++ draft section ???
- [N1717==04-0157] Francis Glassborow and Lois Goldthwaite: explicit class and default definitions (an early proposal).
- Bjarne Stroustrup: Control of class defaults (a dead end).
- [N2326==07-0186] Lawrence Crowl: Defaulted and Deleted Functions.
- [N3174=100164] B. Stroustrup: To move or not to move. An analysis of problems related to generated copy and move operations. Approved.

---

control of defaults: move and copy

By default, a class has 5 operations:

- copy assignment
- copy constructor
- move assignment
- move constructor
- destructor

If you declare any of those you must consider all and explicitly define or default the ones you want. Think of copying, moving, and destruction as closely related operations, rather than individual operations that you can freely mix and match - you can specify arbitrary combinations, but only a few combinations make sense semantically.

If any move, copy, or destructor is explicitly specified (declared, defined, =default, or =delete) by the user, no move is generated by default. If any move, copy, or destructor is explicitly specified (declared, defined, =default, or =delete) by the user, any undeclared copy operations are generated by default, but this is deprecated, so don't rely on that. For example:

```
class X1 {
	X1& operator=(const X1&) = delete;	// Disallow copying
};
```

This implicitly also disallows moving of **X1**s. Copy initialization is allowed, but deprecated.

```
class X2 {
	X2& operator=(const X2&) = delete;
};
```

This implicitly also disallows moving of **X2**s. Copy initialization is allowed, but deprecated.

```
class X3 {
	X3& operator=(X3&&) = delete;	// Disallow moving
};
```

This implicitly also disallows copying of **X3**s.

```
class X4 {
	~X4() = delete;	// Disallow destruction
};
```

This implicitly also disallows moving of **X4**s. Copying is allowed, but deprecated.

I strongly recommend that if you declare one of these five function, you explicitly declare all. For example:

```
template<class T>
class Handle {
	T* p;
public:
	Handle(T* pp) : p{pp} {}
	~Handle() { delete p; }		// user-defined destructor: no implicit copy or move

	Handle(Handle&& h) :p{h.p} { h.p=nullptr; }			// transfer ownership
	Handle& operator=(Handle&& h) { delete p; p=h.p; h.p=nullptr; return *this; }	// transfer ownership

	Handle(const Handle&) = delete;		// no copy
	Handle& operator=(const Handle&) = delete;

	// ...
};
```

See also

- the C++ draft section ???
- [N2326==07-0186] Lawrence Crowl: Defaulted and Deleted Functions.
- [N3174=100164] B. Stroustrup: To move or not to move. An analysis of problems related to generated copy and move operations. Approved.

---

enum class -- scoped and strongly typed enums

The **enum class**es ("new enums", "strong enums") address three problems with traditional C++ enumerations:

- conventional **enum**s implicitly convert to **int**, causing errors when someone does not want an enumeration to act as an integer.
- conventional **enum**s export their enumerators to the surrounding scope, causing name clashes.
- the underlying type of an **enum** cannot be specified, causing confusion, compatibility problems, and makes forward declaration impossible.

enum classs ("strong enums") are strongly typed and scoped:

```
enum Alert { green, yellow, orange, red }; // traditional enum

enum class Color { red, blue };   // scoped and strongly typed enum
                                  // no export of enumerator names into enclosing scope
                                  // no implicit conversion to int
enum class TrafficLight { red, yellow, green };

Alert a = 7;              // error (as ever in C++)
Color c = 7;              // error: no int->Color conversion

int a2 = red;             // ok: Alert->int conversion
int a3 = Alert::red;      // error in C++98; ok in C++11
int a4 = blue;            // error: blue not in scope
int a5 = Color::blue;     // error: not Color->int conversion

Color a6 = Color::blue;   // ok
```

As shown, traditional **enum**s work as usual, but you can now optionally qualify with the **enum**'s name.

The new enums are "enum class" because they combine aspects of traditional enumerations (names values) with aspects of classes (scoped members and absence of conversions).

Being able to specify the underlying type allow simpler interoperability and guaranteed sizes of enumerations:

```
enum class Color : char { red, blue };  // compact representation

enum class TrafficLight { red, yellow, green };  // by default, the underlying type is int

enum E { E1 = 1, E2 = 2, Ebig = 0xFFFFFFF0U };   // how big is an E?
                                                 // (whatever the old rules say;
                                                 // i.e. "implementation defined")

enum EE : unsigned long { EE1 = 1, EE2 = 2, EEbig = 0xFFFFFFF0U };   // now we can be specific
```

It also enables forward declaration of **enum**s:

```
enum class Color_code : char;     // (forward) declaration
void foobar(Color_code* p);       // use of forward declaration
// ...
enum class Color_code : char { red, yellow, green, blue }; // definition
```

The underlying type must be one of the signed or unsigned integer types; the default is **int**.

In the standard library, enum classes are used

- For mapping systems specific error codes: In <system_error>: enum class errc;
- For pointer safety indicators: In <memory>: enum class pointer_safety { relaxed, preferred, strict };
- For I/O stream errors: In <iosfwd>: enum class io_errc { stream = 1 };
- For asynchronous communications error handling: In <future>: enum class future_errc { broken_promise, future_already_retrieved, promise_already_satisfied };

Several of these have operators, such as == defined.

See also

- the C++ draft section 7.2
- [N1513=03-0096] David E. Miller: Improving Enumeration Types (original enum proposal).
- [N2347 = J16/07-0207] David E. Miller, Herb Sutter, and Bjarne Stroustrup: Strongly Typed Enums (revision 3).
- [N2499=08-0009] Alberto Ganesh Barbati: Forward declaration of enumerations.

---

constexpr -- generalized and guaranteed constant expressions

The **constexpr** mechanism

- provides more general constant expressions
- allows constant expressions involving user-defined types
- provides a way to guarantee that an initialization is done at compile time

Consider

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

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

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

Here **constexpr** says that the function must be of a simple form so that it can be evaluated at compile time if given constant expressions arguments.

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

```
constexpr int x1 = bad|eof;     // ok

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

Typically we want the compile-time evaluation guarantee for global or namespace objects, often for objects we want to place in read-only storage.

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

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

constexpr Point origo(0,0);
constexpr int z = origo.x;

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

Please note that **constexpr** is not a general purpose replacement for **const** (or vise versa):

- **const**'s primary function is to express the idea that an object is not modified through an interface (even though the object may very well be modified through other interfaces). It just so happens that declaring an object **const** provides excellent optimization opportunities for the compiler. In particular, if an object is declared **const** and its address isn't taken, a compiler is often able to evaluate its initializer at compile time (though that's not guaranteed) and keep that object in its tables rather than emitting it into the generated code.
- **constexpr**'s primary function is to extend the range of what can be computed at compile time, making such computation type safe. Objects declared **constexpr** have their initializer evaluated at compile time; they are basically values kept in the compiler's tables and only emitted into the generated code if needed.

See also

- the C++ draft 3.6.2 Initialization of non-local objects, 3.9 Types [12], 5.19 Constant expressions, 7.1.5 The constexpr specifier
- [N1521=03-0104] Gabriel Dos Reis: Generalized Constant Expressions (original proposal).
- [N2235=07-0095] Gabriel Dos Reis, Bjarne Stroustrup, and Jens Maurer: Generalized Constant Expressions -- Revision 5.

---

decltype -- the type of an expression

decltype(E) is the type ("declared type") of the name or expression **E** and can be used in declarations. For example:

```
void f(const vector<int>& a, vector<float>& b)
{
        typedef decltype(a[0]*b[0]) Tmp;
        for (int i=0; i<b.size(); ++i) {
                Tmp* p = new Tmp(a[i]*b[i]);
                // ...
        }
        // ...
}
```

This notion has been popular in generic programming under the label "typeof" for a long time, but the **typeof** implementations in actual use were incomplete and incompatible, so the standard version is named **decltype**.

If you just need the type for a variable that you are about to initialize auto is often a simpler choice. You really need **decltype** if you need a type for something that is not a variable, such as a return type.

See also

- the C++ draft 7.1.6.2 Simple type specifiers
- [Str02] Bjarne Stroustrup. Draft proposal for "typeof". C++ reflector message c++std-ext-5364, October 2002. (original suggestion).
- [N1478=03-0061] Jaakko Jarvi, Bjarne Stroustrup, Douglas Gregor, and Jeremy Siek: Decltype and auto (original proposal).
- [N2343=07-0203] Jaakko Jarvi, Bjarne Stroustrup, and Gabriel Dos Reis: Decltype (revision 7): proposed wording.

Initializer lists

Consider

```
vector<double> v = { 1, 2, 3.456, 99.99 };
list<pair<string,string>> languages = {
        {"Nygaard","Simula"}, {"Richards","BCPL"}, {"Ritchie","C"}
};
map<vector<string>,vector<int>> years = {
        { {"Maurice","Vincent", "Wilkes"},{1913, 1945, 1951, 1967, 2000} },
        { {"Martin", "Ritchards"}, {1982, 2003, 2007} },
        { {"David", "John", "Wheeler"}, {1927, 1947, 1951, 2004} }
};
```

Initializer lists are not just for arrays any more. The mechanism for accepting a **{}**-list is a function (often a constructor) accepting an argument of type **std::initializer_list<T>**. For example:

```
void f(initializer_list<int>);
f({1,2});
f({23,345,4567,56789});
f({});  // the empty list
f{1,2}; // error: function call ( ) missing

years.insert({{"Bjarne","Stroustrup"},{1950, 1975, 1985}});
```

The initializer list can be of arbitrary length, but must be homogeneous (all elements must be of a the template argument type, **T**, or convertible to **T**).

A container might implement an initializer-list constructor like this:

```
template<class E> class vector {
public:
        vector (std::initializer_list<E> s) // initializer-list constructor
        {
```

```
                reserve(s.size());      // get the right amount of space
                uninitialized_copy(s.begin(), s.end(), elem);   // initialize elements (in elem[0:s.size()))
                sz = s.size();  // set vector size
        }

        // ... as before ...
};
```

The distinction between direct initialization and copy initialization is maintained for **{}**-initialization, but becomes relevant less frequently because of **{}**-initialization. For example, **std::vector** has an **explicit** constructor from **int** and an initializer-list constructor:

```
vector<double> v1(7);   // ok: v1 has 7 elements
v1 = 9;                 // error: no conversion from int to vector
vector<double> v2 = 9;  // error: no conversion from int to vector

void f(const vector<double>&);
f(9);                           // error: no conversion from int to vector

vector<double> v1{7};           // ok: v1 has 1 element (with its value 7.0)
v1 = {9};                       // ok v1 now has 1 element (with its value 9.0)
vector<double> v2 = {9};        // ok: v2 has 1 element (with its value 9.0)
f({9});                         // ok: f is called with the list { 9 }

vector<vector<double>> vs = {
        vector<double>(10),     // ok: explicit construction (10 elements)
        vector<double>{10},     // ok: explicit construction (1 element with the value 10.0)
        10                      // error: vector's constructor is explicit
};
```

The function can access the **initializer_list** as an immutable sequence. For example:

```
void f(initializer_list<int> args)
{
        for (auto p=args.begin(); p!=args.end(); ++p) cout << *p << "\n";
}
```

A constructor that takes a single argument of type **std::initializer_list** is called an initializer-list constructor.

The standard library containers, **string**, and **regex** have initializer-list constructors, assignment, etc. An initializer-list can be used as a range, e.g. in a range for statement

The initializer lists are part of the scheme for uniform and general initialization.

See also

- the C++ draft 8.5.4 List-initialization [dcl.init.list]
- [N1890=05-0150] Bjarne Stroustrup and Gabriel Dos Reis: Initialization and initializers (an overview of initialization-related problems with suggested solutions).
- [N1919=05-0179] Bjarne Stroustrup and Gabriel Dos Reis: Initializer lists.
- [N2215=07-0075] Bjarne Stroustrup and Gabriel Dos Reis : Initializer lists (Rev. 3) .
- [N2640=08-0150] Jason Merrill and Daveed Vandevoorde: Initializer Lists -- Alternative Mechanism and Rationale (v. 2) (final proposal).

Preventing narrowing

The problem: C and C++ implicitly truncates:

```
int x = 7.3;            // Ouch!
void f(int);
f(7.3);                 // Ouch!
```

However, in C++11, **{}** initialization doesn't narrow:

```
int x0 {7.3};   // error: narrowing
int x1 = {7.3}; // error: narrowing
double d = 7;
int x2{d};              // error: narrowing (double to int)
char x3{7};             // ok: even though 7 is an int, this is not narrowing
vector<int> vi = { 1, 2.3, 4, 5.6 };    // error: double to int narrowing
```

The way C++11 avoids a lot of incompatibilities is by relying on the actual values of initializers (such as 7 in the example above) when it can (and not just type) when deciding what is a narrowing conversion. If a value can be represented exactly as the target type, the conversion is not narrowing.

```
char c1{7};      // OK: 7 is an int, but it fits in a char
char c2{77777};  // error: narrowing (assuming 8-bit chars)
```

Note that floating-point to integer conversions are always considered narrowing -- even 7.0 to 7.

See also

- the C++ draft section 8.5.4.
- [N1890=05-0150] Bjarne Stroustrup and Gabriel Dos Reis: Initialization and initializers (an overview of initialization-related problems with suggested solutions).
- [N2215=07-0075] Bjarne Stroustrup and Gabriel Dos Reis : Initializer lists (Rev. 3) .
- [N2640=08-0150] Jason Merrill and Daveed Vandevoorde: Initializer Lists - Alternative Mechanism and Rationale (v. 2) (primarily on "explicit").

Delegating constructors

In C++98, if you want two constructors to do the same thing, repeat yourself or call "an init() function." For example:

```
class X {
        int a;
        void validate(int x) { if (0<x && x<=max) a=x; else throw bad_X(x); }
public:
        X(int x) { validate(x); }
        X() { validate(42); }
        X(string s) { int x = lexical_cast<int>(s); validate(x); }
        // ...
};
```

Verbosity hinders readability and repetition is error-prone. Both get in the way of maintainability. So, in C++11, we can define one constructor in terms of another:

```
class X {
        int a;
public:
        X(int x) { if (0<x && x<=max) a=x; else throw bad_X(x); }
        X() :X{42} { }
        X(string s) :X{lexical_cast<int>(s)} { }
        // ...
};
```

See also

- the C++ draft section 12.6.2
- N1986==06-0056 Herb Sutter and Francis Glassborow: Delegating Constructors (revision 3).

In-class member initializers

In C++98, only static const members of integral types can be initialized in-class, and the initializer has to be a constant expression. These restrictions ensure that we can do the initialization at compile-time. For example:

```
int var = 7;

class X {
	static const int m1 = 7;        // ok
	const int m2 = 7;                       // error: not static
	static int m3 = 7;              // error: not const
	static const int m4 = var;              // error: initializer not constant expression
	static const string m5 = "odd"; // error: not integral type
	// ...
};
```

The basic idea for C++11 is to allow a non-static data member to be initialized where it is declared (in its class). A constructor can then use the initializer when run-time initialization is needed. Consider:

```
class A {
public:
	int a = 7;
};
```

This is equivalent to:

```
class A {
public:
	int a;
	A() : a(7) {}
};
```

This saves a bit of typing, but the real benefits come in classes with multiple constructors. Often, all constructors use a common initializer for a member:

```
class A {
public:
	A(): a(7), b(5), hash_algorithm("MD5"), s("Constructor run") {}
	A(int a_val) : a(a_val), b(5), hash_algorithm("MD5"), s("Constructor run") {}
	A(D d) : a(7), b(g(d)), hash_algorithm("MD5"), s("Constructor run") {}
	int a, b;
private:
	HashingFunction hash_algorithm;  // Cryptographic hash to be applied to all A instances
	std::string s;                   // String indicating state in object lifecycle
};
```

The fact that **hash_algorithm** and **s** each has a single default is lost in the mess of code and could easily become a problem during maintenance. Instead, we can factor out the initialization of the data members:

```
class A {
public:
	A(): a(7), b(5) {}
	A(int a_val) : a(a_val), b(5) {}
	A(D d) : a(7), b(g(d)) {}
	int a, b;
private:
	HashingFunction hash_algorithm{"MD5"};  // Cryptographic hash to be applied to all A instances
	std::string s{"Constructor run"};       // String indicating state in object lifecycle
};
```

If a member is initialized by both an in-class initializer and a constructor, only the constructor's initialization is done (it "overrides" the default). So we can simplify further:

```
class A {
public:
	A() {}
	A(int a_val) : a(a_val) {}
	A(D d) : b(g(d)) {}
	int a = 7;
	int b = 5;
private:
	HashingFunction hash_algorithm{"MD5"};  // Cryptographic hash to be applied to all A instances
	std::string s{"Constructor run"};       // String indicating state in object lifecycle
    };
```

See also

- the C++ draft section "one or two words all over the place"; see proposal.
- [N2628=08-0138] Michael Spertus and Bill Seymour: Non-static data member initializers.

Inherited constructors

People sometimes are confused about the fact that ordinary scope rules apply to class members. In particular, a member of a base class is not in the same scope as a member of a derived class:

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

struct D : B {
	void f(int);
};

B b;   b.f(4.5);        // fine
D d;   d.f(4.5);        // surprise: calls f(int) with argument 4
```

In C++98, we can "lift" a set of overloaded functions from a base class into a derived class:

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

struct D : B {
	using B::f;     // bring all f()s from B into scope
	void f(int);    // add a new f()
};

B b;   b.f(4.5);        // fine
D d;   d.f(4.5);        // fine: calls D::f(double) which is B::f(double)
```

I have said that "Little more than a historical accident prevents using this to work for a constructor as well as for an ordinary member function." C++11 provides that facility:

```
class Derived : public Base {
public:
	using Base::f;    // lift Base's f into Derived's scope -- works in C++98
	void f(char);     // provide a new f
	void f(int);      // prefer this f to Base::f(int)
```

```
		using Base::Base; // lift Base constructors Derived's scope -- C++11 only
		Derived(char);    // provide a new constructor
		Derived(int);     // prefer this constructor to Base::Base(int)
		// ...
	};
```

If you so choose, you can still shoot yourself in the foot by inheriting constructors in a derived class in which you define new member variables needing initialization:

```
	struct B1 {
		B1(int) { }
	};

	struct D1 : B1 {
		using B1::B1; // implicitly declares D1(int)
		int x;
	};

	void test()
	{
		D1 d(6);	// Oops: d.x is not initialized
		D1 e;		// error: D1 has no default constructor
	}
```

You might remove the bullet from your foot by using a member-initializer:

```
	struct D1 : B1 {
		using B1::B1;	// implicitly declares D1(int)
		int x{0};	// note: x is initialized
	};

	void test()
	{
		D1 d(6);	// d.x is zero
	}
```

See also

- the C++ draft section 12.9.
- [N1890=05-0150] Bjarne Stroustrup and Gabriel Dos Reis: Initialization and initializers (an overview of initialization-related problems with suggested solutions).
- [N1898=05-0158] Michel Michaud and Michael Wong: Forwarding and inherited constructors .
- [N2512=08-0022] Alisdair Meredith, Michael Wong, Jens Maurer: Inheriting Constructors (revision 4).

Static (compile-time) assertions -- static_assert

A static (compile time) assertion consists of a constant expression and a string literal:

```
	static_assert(expression,string);
```

The compiler evaluates the expression and writes the string as an error message if the expression is false (i.e., if the assertion failed). For example:

```
	static_assert(sizeof(long)>=8, "64-bit code generation required for this library.");
	struct S { X m1; Y m2; };
	static_assert(sizeof(S)==sizeof(X)+sizeof(Y),"unexpected padding in S");
```

A **static_assert** can be useful to make assumptions about a program and its treatment by a compiler explicit. Note that since **static_assert** is evaluated at compile time, it cannot be used to check assumptions that depends on run-time values. For example:

```
	int f(int* p, int n)
	{
		static_assert(p==0,"p is not null");	// error: static_assert() expression not a constant expression
		// ...
	}
```

(instead, test and throw an exception in case of failure).

See also

- the C++ draft 7 [4].
- [N1381==02-0039] Robert Klarer and John Maddock: Proposal to Add Static Assertions to the Core Language.
- [N1720==04-0160] Robert Klarer, John Maddock, Beman Dawes, Howard Hinnant: Proposal to Add Static Assertions to the Core Language (Revision 3).

---

long long -- a longer integer

An integer that's at least 64 bits long. For example:

```
	long long x = 9223372036854775807LL;
```

No, there are no **long long long**s nor can **long** be spelled **short long long**.

See also

- the C++ draft ???.
- [05-0071==N1811] J. Stephen Adamczyk: Adding the long long type to C++ (Revision 3).

---

nullptr -- a null pointer literal

nullptr is a literal denoting the null pointer; it is not an integer:

```
	char* p = nullptr;
	int* q = nullptr;
	char* p2 = 0;		// 0 still works and p==p2

	void f(int);
	void f(char*);

	f(0);			// call f(int)
	f(nullptr);		// call f(char*)

	void g(int);
	g(nullptr);		// error: nullptr is not an int
	int i = nullptr;	// error nullptr is not an int
```

See also

- the C++ draft section ???
- [N1488==/03-0071] Herb Sutter and Bjarne Stroustrup: A name for the null pointer: nullptr .
- [N2214 = 07-0074] Herb Sutter and Bjarne Stroustrup: A name for the null pointer: nullptr (revision 4) .

Suffix return type syntax

Consider:

```
template<class T, class U>
??? mul(T x, U y)
{
	return x*y;
}
```

What can we write as the return type? It's "the type of **x*y**", of course, but how can we say that? First idea, use decltype:

```
template<class T, class U>
decltype(x*y) mul(T x, U y) // scope problem!
{
	return x*y;
}
```

That won't work because **x** and **y** are not in scope. However, we can write:

```
template<class T, class U>
decltype(*(T*)(0)**(U*)(0)) mul(T x, U y)	// ugly! and error prone
{
	return x*y;
}
```

However, calling that "not pretty" would be overly polite.

The solution is put the return type where it belongs, after the arguments:

```
template<class T, class U>
auto mul(T x, U y) -> decltype(x*y)
{
	return x*y;
}
```

We use the notation **auto** to mean "return type to be deduced or specified later."

The suffix syntax is not primarily about templates and type deduction, it is really about scope.

```
struct List {
	struct Link { /* ... */ };
	Link* erase(Link* p);	// remove p and return the link before p
	// ...
};

List::Link* List::erase(Link* p) { /* ... */ }
```

The first **List::** is necessary only because the scope of **List** isn't entered until the second **List::**. Better:

```
auto List::erase(Link* p) -> Link* { /* ... */ }
```

Now neither **Link** needs explicit qualification.

See also

- the C++ draft section ???
- [Str02] Bjarne Stroustrup. Draft proposal for "typeof". C++ reflector message c++std-ext-5364, October 2002.
- [N1478=03-0061] Jaakko Jarvi, Bjarne Stroustrup, Douglas Gregor, and Jeremy Siek: Decltype and auto.
- [N2445=07-0315] Jason Merrill: New Function Declarator Syntax Wording.
- [N2825=09-0015] Lawrence Crowl and Alisdair Meredith: Unified Function Syntax.

---

template alias (formerly known as "template typedef")

How can we make a template that's "just like another template" but possibly with a couple of template arguments specified (bound)? Consider:

```
template<class T>
using Vec = std::vector<T,My_alloc<T>>;  // standard vector using my allocator

Vec<int> fib = { 1, 2, 3, 5, 8, 13 }; // allocates elements using My_alloc

vector<int,My_alloc<int>> verbose = fib; // verbose and fib are of the same type
```

The keyword **using** is used to get a linear notation "name followed by what it refers to." We tried with the conventional and convoluted **typedef** solution, but never managed to get a complete and coherent solution until we settled on a less obscure syntax.

Specialization works (you can alias a set of specializations but you cannot specialize an alias) For example:

```
template<int>
struct int_exact_traits {	// idea: int_exact_trait<N>::type is a type with exactly N bits
	typedef int type;
};

template<>
struct int_exact_traits<8> {
	typedef char type;
};

template<>
struct int_exact_traits<16> {
	typedef char[2] type;
};

// ...

template<int N>
using int_exact = typename int_exact_traits<N>::type;  // define alias for convenient notation

int_exact<8> a = 7;	// int_exact<8> is an int with 8 bits
```

In addition to being important in connection with templates, type aliases can also be used as a different (and IMO better) syntax for ordinary type aliases:

```
typedef void (*PFD)(double);	// C style
using PF = void (*)(double);	// using plus C-style type
using P = [](double)->void;	// using plus suffix return type
```

See also

- the C++ draft: 14.6.7 Template aliases; 7.1.3 The typedef specifier
- [N1489=03-0072] Bjarne Stroustrup and Gabriel Dos Reis: Templates aliases for C++.
- [N2258=07-0118] Gabriel Dos Reis and Bjarne Stroustrup: Templates Aliases (Revision 3) (final proposal).

Variadic Templates

Problems to be solved:

- How to construct a class with 1, 2, 3, 4, 5, 6, 7, 8, 9, or ... initializers?
- How to avoid constructing an object out of parts and then copying the result?
- How to construct a tuple?

The last question is the key: Think tuple! If you can make and access general tuples the rest will follow.

Here is an example (from ``A brief introduction to Variadic templates'' (see references)) implementing a general, type-safe, **printf()**. It would probably be better to use **boost::format**, but consider:

```
const string pi = "pi";
const char* m = "The value of %s is about %g (unless you live in %s).\n";
printf(m,pi,3.14159,"Indiana");
```

The simplest case of **printf()** is when there are no arguments except the format string, so we'll handle that first:

```
void printf(const char* s)
{
	while (s && *s) {
		if (*s=='%' && *++s!='%')	// make sure that there wasn't meant to be more arguments
						// %% represents plain % in a format string
		 throw runtime_error("invalid format: missing arguments");
		std::cout << *s++;
	}
}
```

That done, we must handle **printf()** with more arguments:

```
template<typename T, typename... Args>		// note the "..."
void printf(const char* s, T value, Args... args)	// note the "..."
{
	while (s && *s) {
		if (*s=='%' && *++s!='%') {	// a format specifier (ignore which one it is)
			std::cout << value;		// use first non-format argument
			return printf(++s, args...); 	// ``peel off'' first argument
		}
		std::cout << *s++;
	}
	throw std::runtime error("extra arguments provided to printf");
}
```

This code simply ``peels off'' the first non-format argument and then calls itself recursively. When there are no more non-format arguments, it calls the first (simpler) **printf()** (above). This is rather standard functional programming done at compile time. Note how the overloading of **<<** replaces the use of the (possibly erroneous) ``hint'' in the format specifier.

The **Args...**defines what is called a ``parameter pack.'' That's basically a sequence of (type/value) pairs from which you can ``peel off'' arguments starting with the first. When **printf()** is called with one argument, the first definition (**printf(const char*)**) is chosen. When **printf()** is called with two or more arguments, the second definition (**printf(const char*, T value, Args... args)**) is chosen, with the first argument as **s**, the second as **value**, and the rest (if any) bundled into the parameter pack **args** for later use. In the call

```
printf(++s, args...);
```

The parameter pack **args** is expanded so that the next argument can now be selected as **value**. This carries on until **args** is empty (so that the first **printf()** is called).

If you are familiar with functional programming, you should find this an unusual notation for a pretty standard technique. If not, here are some small technical examples that might help. First we can declare and use a simple variadic template function (just like **printf()** above):

```
template<class ... Types>
	void f(Types ... args); // variadic template function
				// (i.e. a function that can take an arbitrary number of arguments of arbitrary types)
f();		// OK: args contains no arguments
f(1);		// OK: args contains one argument: int
f(2, 1.0);	// OK: args contains two arguments: int and double
```

We can build a variadic type:

```
template<typename Head, typename... Tail>
class tuple<Head, Tail...>
	: private tuple<Tail...> {	// here is the recursion
			// Basically, a tuple stores its head (first (type/value) pair
			// and derives from the tuple of its tail (the rest of the (type/value) pairs.
			// Note that the type is encoded in the type, not stored as data
	typedef tuple<Tail...> inherited;
public:
	tuple() { }	// default: the empty tuple

	// Construct tuple from separate arguments:
	tuple(typename add_const_reference<Head>::type v, typename add_const_reference<Tail>::type... vtail)
		: m_head(v), inherited(vtail...) { }

	// Construct tuple from another tuple:
	template<typename... VValues>
	tuple(const tuple<VValues...>& other)
	:	m_head(other.head()), inherited(other.tail()) { }

	template<typename... VValues>
	tuple& operator=(const tuple<VValues...>& other)	// assignment
	{
		m_head = other.head();
		tail() = other.tail();
		return *this;
	}

	typename add_reference<Head>::type head() { return m_head; }
	typename add_reference<const Head>::type head() const { return m_head; }

	inherited& tail() { return *this; }
	const inherited& tail() const { return *this; }
protected:
	Head m_head;
}
```

Given that definition, we can make tuples (and copy and manipulate them):

```
tuple<string,vector<int>,double> tt("hello",{1,2,3,4},1.2);
string h = tt.head();	// "hello"
tuple<vector<int>,double> t2 = tt.tail();	// {{1,2,3,4},1.2};
```

It can get a bit tedious to mention all of those types, so often, we deduce them from argument types, e.g. using the standard library **make_tuple()**:

```
template<class... Types>
tuple<Types...> make_tuple(Types&&... t)	// this definition is somewhat simplified (see standard 20.5.2.2)
{
	return tuple<Types...>(t...);
}

string s = "Hello";
```

```
vector<int> v = {1,22,3,4,5};
auto x = make_tuple(s,v,1.2);
```

See also:

- Standard 14.6.3 Variadic templates
- [N2151==07-0011] D. Gregor, J. Jarvi: Variadic Templates for the C++0x Standard Library.
- [N2080==06-0150] D. Gregor, J. Jarvi, G. Powell: Variadic Templates (Revision 3).
- [N2087==06-0157] Douglas Gregor: A Brief Introduction to Variadic Templates.
- [N2772==08-0282] L. Joly, R. Klarer: Variadic functions: Variadic templates or initializer lists? -- Revision 1.
- [N2551==08-0061] Sylvain Pion: A Variadic std::min(T, ...) for the C++ Standard Library (Revision 2) .
- Anthony Williams: An Introduction to Variadic Templates in C++0x. DevX.com, May 2009.

Uniform initialization syntax and semantics

C++ offers several ways of initializing an object depending on its type and the initialization context. When misused, the error can be surprising and the error messages obscure. Consider:

```
string a[] = { "foo", " bar" };          // ok: initialize array variable
vector<string> v = { "foo", " bar" };    // error: initializer list for non-aggregate vector
void f(string a[]);
f( { "foo", " bar" } );                  // syntax error: block as argument
```

and

```
int a = 2;              // ``assignment style''
int aa[] = { 2, 3 };    // assignment style with list
complex z(1,2);         // ``functional style'' initialization
x = Ptr(y);             // ``functional style'' for conversion/cast/construction
```

and

```
int a(1);       // variable definition
int b();        // function declaration
int b(foo);     // variable definition or function declaration
```

It can be hard to remember the rules for initialization and to choose the best way.

The C++11 solution is to allow **{}**-initializer lists for all initialization:

```
X x1 = X{1,2};
X x2 = {1,2};   // the = is optional
X x3{1,2};
X* p = new X{1,2};

struct D : X {
        D(int x, int y) :X{x,y} { /* ... */ };
};

struct S {
        int a[3];
        S(int x, int y, int z) :a{x,y,z} { /* ... */ }; // solution to old problem
};
```

Importantly, **X{a}** constructs the same value in every context, so that **{}**-initialization gives the same result in all places where it is legal. For example:

```
X x{a};
X* p = new X{a};
z = X{a};         // use as cast
f({a});           // function argument (of type X)
return {a};       // function return value (function returning X)
```

See also

- the C++ draft section ???
- [N2215==07-0075] Bjarne Stroustrup and Gabriel Dos Reis: Initializer lists (Rev. 3) .
- [N2640==08-0150] Jason Merrill and Daveed Vandevoorde: Initializer Lists -- Alternative Mechanism and Rationale (v. 2) (final proposal).

Rvalue references

The distinction between lvalues (what can be used on the left-hand side of an assignment) and rvalues (what can be used on the right-hand side of an assignment) goes back to Christopher Strachey (the father of C++'s distant ancestor CPL and of denotational semantics). In C++, non-const references can bind to lvalues and const references can bind to lvalues or rvalues, but there is nothing that can bind to a non-const rvalue. That's to protect people from changing the values of temporaries that are destroyed before their new value can be used. For example:

```
void incr(int& a) { ++a; }
int i = 0;
incr(i);        // i becomes 1
incr(0);        // error: 0 is not an lvalue
```

If that **incr(0)** were allowed either some temporary that nobody ever saw would be incremented or - far worse - the value of 0 would become 1. The latter sounds silly, but there was actually a bug like that in early Fortran compilers that set aside a memory location to hold the value 0.

So far, so good, but consider

```
template<class T> swap(T& a, T& b)             // "old style swap"
{
        T tmp(a);       // now we have two copies of a
        a = b;          // now we have two copies of b
        b = tmp;        // now we have two copies of tmp (aka a)
}
```

If **T** is a type for which it can be expensive to copy elements, such as string and vector, swap becomes an expensive operation (for the standard library, we have specializations of string and vector **swap()** to deal with that). Note something curious: We didn't want any copies at all. We just wanted to move the values of **a**, **b**, and **tmp** around a bit.

In C++11, we can define "move constructors" and "move assignments" to move rather than copy their argument:

```
template<class T> class vector {
        // ...
        vector(const vector&);                  // copy constructor
        vector(vector&&);                       // move constructor
        vector& operator=(const vector&);       // copy assignment
        vector& operator=(vector&&);            // move assignment
};      // note: move constructor and move assignment takes non-const &&
        // they can, and usually do, write to their argument
```

The && indicates an "rvalue reference". An rvalue reference can bind to an rvalue (but not to an lvalue):

```
X a;
X f();
```

```
X& r1 = a;              // bind r1 to a (an lvalue)
X& r2 = f();            // error: f() is an rvalue; can't bind

X&& rr1 = f();  // fine: bind rr1 to temporary
X&& rr2 = a;    // error: bind a is an lvalue
```

The idea behind a move assignment is that instead of making a copy, it simply takes the representation from its source and replaces it with a cheap default. For example, for strings **s1=s2** using the move assignment would not make a copy of **s2**'s characters; instead, it would just let **s1** treat those characters as its own and somehow delete **s1**'s old characters (maybe by leaving them in **s2**, which presumably are just about to be destroyed).

How do we know whether it's ok to simply move from a source? We tell the compiler:

```
template<class T>
void swap(T& a, T& b)   // "perfect swap" (almost)
{
        T tmp = move(a);        // could invalidate a
        a = move(b);            // could invalidate b
        b = move(tmp);          // could invalidate tmp
}
```

move(x) means "you can treat **x** as an rvalue". Maybe it would have been better if **move()** had been called **rval()**, but by now **move()** has been used for years. The **move()** template function can be written in C++11 (see the "brief introduction") and uses rvalue references.

Rvalue references can also be used to provide perfect forwarding.

In the C++11 standard library, all containers are provided with move constructors and move assignment and operations that insert new elements, such as **insert()** and **push_back()** have versions that take rvalue references. The net result is that the standard containers and algorithms quietly - without user intervention - improve in performance because they copy less.

See also

- the C++ draft section ???
- N1385 N1690 N1770 N1855 N1952
- [N2027==06-0097] Howard Hinnant, Bjarne Stroustrup, and Bronek Kozicki: A brief introduction to rvalue references
- [N1377=02-0035] Howard E. Hinnant, Peter Dimov, and Dave Abrahams: A Proposal to Add Move Semantics Support to the C++ Language (original proposal).
- [N2118=06-0188] Howard Hinnant: A Proposal to Add an Rvalue Reference to the C++ Language Proposed Wording (Revision 3) (final proposal).

---

unions (generalized)

In C++98 (as in the earlier versions of C++), a member with a user-defined constructor, destructor, or assignment cannot be a member of a union:

```
union U {
        int m1;
        complex<double> m2;     // error (silly): complex has constructor
        string m3;              // error (not silly): string has a serious invariant
                        // maintained by ctor, copy, and dtor
};
```

In particular

```
U u;                    // which constructor, if any?
u.m1 = 1;               // assign to int member
string s = u.m3;        // disaster: read from string member
```

Obviously, it's illegal to write one member and then read another but people do that nevertheless (usually by mistake).

C++11 modifies the restrictions of unions to make more member types feasible; in particular, it allows a member of types with constructors and destructors. It also adds a restriction to make the more flexible unions less error-prone by encouraging the building of discriminated unions.

Union member types are restricted:

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

For example:

```
union U1 {
        int m1;
        complex<double> m2;     // ok
};

union U2 {
        int m1;
        string m3;      // ok
};
```

This may look error-prone, but the new restriction helps. In particular:

```
U1 u;           // ok
u.m2 = {1,2};   // ok: assign to the complex member
U2 u2;          // error: the string destructor caused the U2 destructor to be deleted
U2 u3 = u2;     // error: the string copy constructor caused the U2 copy constructor to be deleted
```

Basically, **U2** is useless unless you embed it in a struct that keeps track of which member (variant) is used. So, build discriminate unions, such as:

```
class Widget {  // Three alternative implementations represented as a union
private:
        enum class Tag { point, number, text } type;    // discriminant
        union {         // representation
                point p;          // point has constructor
                int i;
                string s;        // string has default constructor, copy operations, and destructor
        };
        // ...
        Widget& operator=(const Widget& w)      // necessary because of  the string variant
        {
                if (type==Tag::text && w.type==Tag::text) {
                        s = w.s;                // usual string assignment
                        return *this;
                }

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

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

See also:

- li>the C++ draft section 9.5
- [N2544=08-0054] Alan Talbot, Lois Goldthwaite, Lawrence Crowl, and Jens Maurer: Unrestricted unions (Revision 2)

PODs (generalized)

A POD ("Plain Old Data") is something that can be manipulated like a C struct, e.g. copies with **memcpy()**, initializes with **memset()**, etc. In C++98 the actual definition of POD is based on a set of restrictions on the use of language features used in the definition of a struct:

```
struct S { int a; };    // S is a POD
struct SS { int a; SS(int aa) : a(aa) { } }; // SS is not a POD
struct SSS { virtual void f(); /* ... */ };
```

In C++11, S and SS are "standard layout types" (a.k.a. POD) because there is really nothing "magic" about SS: the constructor does not affect the layout (so memcpy() would be fine), only the initialization rules (memset() would be bad - not enforcing the invariant). However, SSS will still have an embedded vptr and will not be anything like "plain old data." C++11 defines POD, trivially copyable types, trivial types, and standard-layout types to deal with various technical aspects of what used to be PODs. POD is defined recursively

- If all your members and bases are PODs, you're a POD
- As usual (details in section 9 [10])
  - No virtual functions
  - No virtual bases
  - No references
  - No multiple access specifiers

The most important aspect of C++11 PODs are that adding or subtracting constructors do not affect layout or performance.

See also:

- the C++ draft section 3.9 and 9 [10]
- [N2294=07-0154] Beman Dawes: POD's Revisited; Resolving Core Issue 568 (Revision 4).

Raw string literals

In many cases, such as when you are writing regular expressions for the use with the standard regex library, the fact that a backslash (\) is an escape character is a real nuisance (because in regular expressions backslash is used to introduce special characters representing character classes). Consider how to write the pattern representing two words separated by a backslash (**\w\\\w**):

```
string s = "\\w\\\\\\w";        // I hope I got that right
```

Note that the backslash character is represented as two backslashes in a regular expression. Basically, a "raw string literal" is a string literal where a backslash is just a backslash so that our example becomes:

```
string s = R"(\w\\\w)"; // I'm pretty sure I got that right
```

The original proposal for raw strings presents this as a motivating example

```
"('(?:[^\\\\']|\\\\.)*'|\"(?:[^\\\\\"]|\\\\.)*\")|" // Are the five backslashes correct or not?
                                                    // Even experts become easily confused.
```

The **R"(...)"** notation is a bit more verbose than the "plain" **"..."** but "something more" is necessary when you don't have an escape character: How do you put a quote in a raw string? Easy, unless it is preceded by a **)**:

```
R"("quoted string")"    // the string is "quoted string"
```

So, how do we get the character sequence **)"** into a raw string? Fortunately, that's a rare problem, but **"(...)"** is only the default delimiter pair. We can add delimiters before and after the **(...)** in **"(...)"**. For example

```
R"***("quoted string containing the usual terminator (")")***"  // the string is "quoted string containing the usual terminator (")"
```

The character sequence after **)** must be identical to the sequence before the **(**. This way we can cope with (almost) arbitrarily complicated patterns.

The initial **R** of a raw string can be preceded by an encoding-prefix: **u8**, **u**, **U**, or **L**. For example **u8R"(fdfdfa)"** is an UTF-8 string literal.

See

- Standard 2.13.4
- [N2053=06-0123] Beman Dawes: Raw string literals. (original proposal)
- [N2442=07-0312] Lawrence Crowl and Beman Dawes: Raw and Unicode String Literals; Unified Proposal (Rev. 2). (final proposal combined with the User-defined literals proposal).
- [N3077==10-0067] Jason Merrill: Alternative approach to Raw String issues. (replacing **[** with **(**);

User-defined literals

C++ provides literals for a variety of built-in types (2.14 Literals):

```
123     // int
1.2     // double
1.2F    // float
'a'     // char
1ULL    // unsigned long long
0xD0    // hexadecimal unsigned
"as"    // string
```

However, in C++98 there are no literals for user-defined types. This can be a bother and also seen as a violation of the principle that user-defined types should be supported as well as built-in types are. In particular, people have requested:

```
"Hi!"s                  // string, not ``zero-terminated array of char''
1.2i                    // imaginary
123.4567891234df        // decimal floating point (IBM)
101010111000101b        // binary
123s                    // seconds
123.56km                // not miles! (units)
1234567890123456789012345678901234567890x       // extended-precision
```

C++11 supports ``user-defined literals'' through the notion of *literal operator*s that map literals with a given suffix into a desired type. For example:

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

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

Note the use of **constexpr** to enable compile-time evaluation. Given those, we can write

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

auto z = 2+1i;  // complex(2,1)
```

The basic (implementation) idea is that after parsing what could be a literal, the compiler always check for a suffix. The user-defined literal mechanism simply allows the user to specify a new suffix and what is to be done with the literal before it. It is not possible to redefine the meaning of a built-in literal suffix or augment the syntax of literals. A literal operator can request to get its (preceding) literal passed ``cooked'' (with the value it would have had if the new suffix hadn't been defined) or ``uncooked'' (as a string).

To get an ``uncooked'' string, simply request a single **const char*** argument:

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

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

Here the C-style string **"1234567890123456789012345678901234567890"** is passed to **operator"" x()**. Note that we did not explicitly put those digits into a string.

There are four kinds of literals that can be suffixed to make a user-defined literal

- integer literal: accepted by a literal operator taking a single **unsigned long long** or **const char*** argument.
- floating-point literal: accepted by a literal operator taking a single **long double** or **const char*** argument.
- string literal: accepted by a literal operator taking a pair of **(const char*, size_t)** arguments.
- character literal: accepted by a literal operator taking a single **char** argument.

Note that you cannot make a literal operator for a string literal that takes just a **const char*** argument (and no size). For example:

```
string operator"" S(const char* p);            // warning: this will not work as expected

"one two"S;     // error: no applicable literal operator
```

The rationale is that if we want to have ``a different kind of string'' we almost always want to know the number of characters anyway.

Suffixes will tend to be short (e.g. **s** for string, **i** for imaginary, **m** for meter, and **x** for extended), so different uses could easily clash. Use namespaces to prevent clashes:

```
namespace Numerics {
        // ...
        class Bignum { /* ... */ };
        namespace literals {
                operator"" X(char const*);
        }
}

using namespace Numerics::literals;
```

See also:

- Standard 2.14.8 User-defined literals
- [N2378==07-0238] Ian McIntosh, Michael Wong, Raymond Mak, Robert Klarer, Jens Mauer, Alisdair Meredith, Bjarne Stroustrup, David Vandevoorde: User-defined Literals (aka. Extensible Literals (revision 3)).

Attributes

``Attributes'' is a new standard syntax aimed at providing some order in the mess of facilities for adding optional and/or vendor specific information into source code (e.g. **__attribute__**, **__declspec**, and **#pragma**). C++11 attributes differ from existing syntaxes by being applicable essentially everywhere in code and always relating to the immediately preceding syntactic entity. For example:

```
void f [[ noreturn ]] ()        // f() will never return
{
        throw "error";  // OK
}

struct foo* f [[carries_dependency]] (int i);   // hint to optimizer
int* g(int* x, int* y [[carries_dependency]]);
```

As you can see, an attribute is placed within double square brackets: **[[...]]**. **noreturn** and **carries_dependency** are the two attributes defined in the standard.

There is a reasonable fear that attributes will be used to create language dialects. The recommendation is to use attributes to only control things that do not affect the meaning of a program but might help detect errors (e.g. **[[noreturn]]**) or help optimizers (e.g. **[[carries_dependency]]**).

One planned use for attributes is improved support for OpenMP. For example:

```
for [[omp::parallel()]] (int i=0; i<v.size(); ++i) {
        // ...
}
```

As shown, attributes can be qualified.

See also:

- Standard: 7.6.1 Attribute syntax and semantics, 7.6.3-4 noreturn, carries_dependency 8 Declarators, 9 Classes, 10 Derived classes, 12.3.2 Conversion functions
- [N2418=07-027] Jens Maurer, Michael Wong: Towards support for attributes in C++ (Revision 3)

Lambdas

A lambda expression is a mechanism for specifying a function object. The primary use for a lambda is to specify a simple action to be performed by some function. For example:

```
vector<int> v = {50, -10, 20, -30};

std::sort(v.begin(), v.end());  // the default sort
// now v should be { -30, -10, 20, 50 }

// sort by absolute value:
std::sort(v.begin(), v.end(), [](int a, int b) { return abs(a)<abs(b); });
// now v should be { -10, 20, -30, 50 }
```

The argument **[](int a, int b) { return abs(a)<abs(b); }** is a "lambda" (or "lambda function" or "lambda expression"), which specifies an operation that given two integer arguments **a** and **b** returns the result of comparing their absolute values.

A lambda expression can access local variables in the scope in which it is used. For example:

```
void f(vector<Record>& v)
{
        vector<int> indices(v.size());
        int count = 0;
        generate(indices.begin(),indices.end(),[&count](){ return count++; });

        // sort indices in the order determined by the name field of the records:
        std::sort(indices.begin(), indices.end(), [&](int a, int b) { return v[a].name<v[b].name; });
        // ...
}
```

Some consider this "really neat!"; others see it as a way to write dangerously obscure code. IMO, both are right.

The **[&]** is a "capture list" specifying that local names used will be passed by reference. We could have said that we wanted to "capture" only **v**, we could have said so: **[&v]**. Had we wanted to pass **v** by value, we could have said so: **[=v]**. Capture nothing is **[]**, capture all by references is **[&]**, and capture all by value is **[=]**.

If an action is neither common nor simple, I recommend using a named function object or function. For example, the example above could have been written:

```
void f(vector<Record>& v)
{
        vector<int> indices(v.size());
        int count = 0;
        generate(indices.begin(),indices.end(),[&](){ return ++count; });

        struct Cmp_names {
                const vector<Record>& vr;
                Cmp_names(const vector<Record>& r) :vr(r) { }
                bool operator()(int a, int b) const { return vr[a].name<vr[b].name; }
        };

        // sort indices in the order determined by the name field of the records:
        std::sort(indices.begin(), indices.end(), Cmp_names(v));
        // ...
}
```

For a tiny function, such as this Record name field comparison, the function object notation is verbose, though the generated code is likely to be identical. In C++98, such function objects had to be non-local to be used as template argument; in C++11 this is no longer necessary.

To specify a lambda you must provide

- its capture list: the list of variables it can use (in addition to its arguments), if any (**[&]** meaning "all local variables passed by reference" in the Record comparison example). If no names needs to be captured, a lambda starts with plain **[]**.
- (optionally) its arguments and their types (e.g, **(int a, int b)**)
- The action to be performed as a block (e.g., **{ return v[a].name<v[b].name; }**).
- (optionally) the return type using the new suffix return type syntax; but typically we just deduce the return type from the return statement. If no value is returned **void** is deduced.

See also:

- Standard 5.1.2 Lambda expressions
- [N1968=06-0038] Jeremiah Willcock, Jaakko Jarvi, Doug Gregor, Bjarne Stroustrup, and Andrew Lumsdaine: Lambda expressions and closures for C++ (original proposal with a different syntax)
- [N2550=08-0060] Jaakko Jarvi, John Freeman, and Lawrence Crowl: Lambda Expressions and Closures: Wording for Monomorphic Lambdas (Revision 4) (final proposal).
- [N2859=09-0049] Daveed Vandevoorde: New wording for C++0x Lambdas.

Local types as template arguments

In C++98, local and unnamed types could not be used as template arguments. This could be a burden, so C++11 lifts the restriction:

```
void f(vector<X>& v)
{
        struct Less {
                bool operator()(const X& a, const X& b) { return a.v<b.v; }
        };
        sort(v.begin(), v.end(), Less());       // C++98: error: Less is local
                                                // C++11: ok
}
```

In C++11, we also have the alternative of using a lambda expression:

```
void f(vector<X>& v)
{
        sort(v.begin(), v.end(),
              [] (const X& a, const X& b) { return a.v<b.v; }); // C++11
}
```

It is worth remembering that naming action can be quite useful for documentation and an encouragement to good design. Also, non-local (necessarily named) entities can be reused.

C++11 also allows values of unnamed types to be used as template arguments:

```
template<typename T> void foo(T const& t){}
enum X { x };
enum { y };

int main()
{
        foo(x);         // C++98: ok; C++11: ok
        foo(y);         // C++98: error; C++11: ok
        enum Z { z };
        foo(z);         // C++98: error; C++11: ok
}
```

See also:

- Standard: Not yet: CWG issue 757
- [N2402=07-0262] Anthony Williams: Names, Linkage, and Templates (rev 2).
- [N2657] John Spicer: Local and Unnamed Types as Template Arguments.

noexcept -- preventing exception propagation

If a function cannot throw an exception or if the program isn't written to handle exceptions thrown by a function, that function can be declared **noexcept**. For example:

```
extern "C" double sqrt(double) noexcept;        // will never throw

vector<double> my_computation(const vector<double>& v) noexcept // I'm not prepared to handle memory exhaustion
{
        vector<double> res(v.size());   // might throw
        for(int i; i<v.size(); ++i) res[i] = sqrt(v[i]);
        return res;
}
```

If a function declared **noexcept** throws (so that the exception tries to escape, the **noexcept** function) the program is terminated (by a call to **terminate()**). The call of **terminate()** cannot rely on objects being in well-defined states (i.e. there is no guarantees that destructors have been invoked, no guaranteed stack unwinding, and no possibility for resuming the program as if no problem had been encountered). This is deliberate and makes **noexcept** a simple, crude, and very efficient mechanism (much more efficient than the old dynamic **throw()** mechanism).

It is possibly to make a function conditionally **noexcept**. For example, an algorithm can be specified to be **noexcept** if (and only if) the operations it uses on a template argument are **noexcept**:

```
	template<class T>
	void do_f(vector<T>& v) noexcept(noexcept(f(v.at(0)))) // can throw if f(v.at(0)) can
	{
		for(int i; i<v.size(); ++i)
			v.at(i) = f(v.at(i));
	}
```

Here, I first use **noexcept** as an operator: **noexcept(f(v.at(0)))** is true if **f(v.at(0))** can't throw, that is if the **f()** and **at()** used are **noexcept**.

The **noexcept()** operator is a constant expression and does not evaluate its operand.

The general form of a **noexcept** declaration is **noexcept(expression)** and ``plain **noexcept**'' is simply a shorthand for **noexcept(true)**. All declarations of a function must have compatible **noexcept** specifications.

A destructor shouldn't throw; a generated destructor is implicitly **noexcept** (independently of what code is in its body) if all of the members of its class have **noexcept** destructors.

It is typically a bad idea to have a move operation throw, so declare those **noexcept** wherever possible. A generated copy or move operation is implicitly **noexcept** if all of the copy or move operations it uses on members of its class have **noexcept** destructors.

noexcept is widely and systematically used in the standard library to improve performance and clarify requirements. See also:

- Standard: 15.4 Exception specifications [except.spec].
- Standard: 5.3.7 noexcept operator [expr.unary.noexcept].
- [N3103==10-0093] D. Kohlbrenner, D. Svoboda, and A. Wesie: Security impact of noexcept. (Noexcept **must** terminate, as it does).
- [N3167==10-0157] David Svoboda: Delete operators default to noexcept.
- [N3204==10-0194] Jens Maurer: Deducing "noexcept" for destructors
- [N3050==10-0040] D. Abrahams, R. Sharoni, and D. Gregor: Allowing Move Constructors to Throw (Rev. 1).

---

alignment

Occasionally, especially when we are writing code that manipulate raw memory, we need to specify a desired alignment for some allocation. For example:

```
	alignas(double) unsigned char c[1024];	 // array of characters, suitably aligned for doubles
	alignas(16) char[100];			// align on 16 byte boundary
```

There is also an **alignof** operator that returns the alignment of its argument (which must be a type). For example

```
	constexpr int n = alignof(int);		// ints are aligned on n byte boundaries
```

See also:

- Standard: 5.3.6 Alignof [expr.alignof]
- Standard: 7.6.2 Alignment specifier [dcl.align]
- [N3093==10-0083] Lawrence Crowl: C and C++ Alignment Compatibility. Aligning the proposal to C's later proposal.
- [N1877==05-0137] Attila (Farkas) Feh�r: Adding Alignment Support to the C++ Programming Language. The original proposal.

Override controls: override

No special keyword or annotation is needed for a function in a derived class to override a function in a base class. For example:

```
	struct B {
		virtual void f();
		virtual void g() const;
		virtual void h(char);
		void k();	// not virtual
	};

	struct D : B {
		void f();	// overrides B::f()
		void g();	// doesn't override B::g() (wrong type)
		virtual void h(char);	// overrides B::h()
		void k();	// doesn't override B::k() (B::k() is not virtual)
	};
```

This can cause confusion (what did the programmer mean?), and problems if a compiler doesn't warn against suspicious code. For example,

- Did the programmer mean to override **B::g()**? (almost certainly yes).
- Did the programming mean to override **B::h(char)**? (probably not because of the redundant explicit **virtual**).
- Did the programmer mean to override **B::k()**? (probably, but that's not possible).

To allow the programmer to be more explicit about overriding, we now have the "contextual keyword" **override**:

```
	struct D : B {
		void f() override;	// OK: overrides B::f()
		void g() override;	// error: wrong type
		virtual void h(char);	// overrides B::h(); likely warning
		void k() override;	// error: B::k() is not virtual
	};
```

A declaration marked **override** is only valid if there is a function to override. The problem with **h()** is not guaranteed to be caught (because it is not an error according to the language definition) but it is easily diagnosed.

override is only a *contextual* keyword, so you can still use it as an identifier:

```
int override = 7;	// not recommended
```

See also:

- Standard: 10 Derived classes [class.derived] [9]
- Standard: 10.3 Virtual functions [class.virtual]
- [N3234==11-0004] Ville Voutilainen: Remove explicit from class-head.
- [N3151==10-0141] Ville Voutilainen: Keywords for override control. Earlier, more elaborate design.
- [N3163==10-0153] Herb Sutter: Override Control Using Contextual Keywords. Alternative earlier more elaborate design.
- [N2852==09-0042] V. Voutilainen, A. Meredith, J. Maurer, and C. Uzdavinis: Explicit Virtual Overrides. Earlier design based on attributes.
- [N1827==05-0087] C. Uzdavinis and A. Meredith: An Explicit Override Syntax for C++. The original proposal.

Override controls: final

Sometimes, a programmer wants to prevent a virtual function from being overridden. This can be achieved by adding the specifier **final**. For example:

```
struct B {
	virtual void f() const final;	// do not override
	virtual void g();
};

struct D : B {
	void f() const;		// error: D::f attempts to override final B::f
	void g();		// OK
};
```

There are legitimate reasons for wanting to prevent overriding, but I'm afraid that most examples I have been shown to demonstrate the need for **final** have been based on mistaken assumptions on how expensive **virtual** functions are (usually based on experience with other languages). So, if you feel the urge to add a **final** specifier, please double check that the reason is logical: Would semantic errors be likely if someone defined a class that overwrote that virtual function? Adding **final** closes the possibility of a future user of the class might provide a better implementation of the function for some class you haven't thought of. If you don't want to keep that option open, why did you define the function to be **virtual** in the first place? Most reasonable answers to that question that I have encountered have been along the lines: This is a fundamental function in a framework that the framework builders needed to override but isn't safe for general users to override. My bias is to be suspicious towards such claims.

If it is performance (inlining) you want or you simply never want to override, it is typically better not to define a function to be **virtual** in the first place. This is not Java.

final is only a *contextual* keyword, so you can still use it as an identifier:

```
int final = 7;	// not recommended
```

See also:

- Standard: 10 Derived classes [class.derived] [9]
- Standard: 10.3 Virtual functions [class.virtual]

C99 features

To preserve a high degree of compatibility, a few minor changes to the language were introduced in collaboration with the C standards committee:

- long long.
- Extended integral types (i.e. rules for optional longer int types).
- UCN changes [N2170==07-0030] ``lift the prohibitions on control and basic source universal character names within character and string literals.''
- concatenation of narrow/wide strings.
- ***Not*** VLAs (Variable Length Arrays; thank heaven for small mercies).

Some extensions of the preprocessing rules were added:

- **__func__** a macro that expands to the name of the lexically current function
- **__STDC_HOSTED__**
- **_Pragma**: **_Pragma(X)** expands to **#pragma X**
- vararg macros (overloading of macros with different number of arguments)

```
	#define report(test, ...) ((test)?puts(#test):printf(_ _VA_ARGS_ _))
```

- empty macro arguments

A lot of standard library facilities were inherited from C99 (essentially all changes to the C99 library from its C89 predecessor):

See:

- Standard: 16.3 Macro replacement.
- [N1568=04-0008] P.J. Plauger: PROPOSED ADDITIONS TO TR-1 TO IMPROVE COMPATIBILITY WITH C99.

Extended integer types

There are a set of rules for how an extended (precision) integer type should behave if one exists.

See

- [06-0058==N1988] J. Stephen Adamczyk: Adding extended integer types to C++ (Revision 1).

Dynamic Initialization and Destruction with Concurrency

Sorry, I have not had time to write this entry. See

- [N2660 = 08-0170] Lawrence Crowl: Dynamic Initialization and Destruction with Concurrency (Final proposal).

---

thread-local storage (thread_local)

Sorry, I have not had time to write this entry. See

- [N2659 = 08-0169] Lawrence Crowl: Thread-Local Storage (Final proposal).

Unicode characters

Sorry, I have not had time to write this entry. Please come back later.

- ?

Copying and rethrowing exceptions

How do you catch an exception and then rethrow it on another thread? Use a bit of library magic as described in the standard 18.8.5 Exception Propagation:

- **exception_ptr current_exception();** *Returns: An exception_ptr object that refers to the currently handled exception (15.3) or a copy of the currently handled exception, or a null exception_ptr object if no exception is being handled. The referenced object shall remain valid at least as long as there is an exception_ptr object that refers to it. ...*
- **void rethrow_exception(exception_ptr p);**
- **template<class E> exception_ptr copy_exception(E e);** *Effects: as if*

```
	try {
		throw e;
	} catch(...) {
		return current_exception();
	}
```

  This is particularly useful for transmitting an exception from one thread to another

Extern templates

A template specialization can be explicitly declared as a way to suppress multiple instantiations. For example:

```
#include "MyVector.h"

extern template class MyVector<int>; // Suppresses implicit instantiation below --
                                // MyVector<int> will be explicitly instantiated elsewhere

void foo(MyVector<int>& v)
{
        // use the vector in here
}
```

The ``elsewhere'' might look something like this:

```
#include "MyVector.h"

template class MyVector<int>; // Make MyVector available to clients (e.g., of the shared library)
```

This is basically a way of avoiding significant redundant work by the compiler and linker.

See

- Standard 14.7.2 Explicit instantiation
- [N1448==03-0031] Mat Marcus and Gabriel Dos Reis: Controlling Implicit Template Instantiation.

Inline namespace

The **inline namespace** mechanism is intended to support library evolution by providing a mechanism that support a form of versioning. Consider:

```
// file V99.h:
inline namespace V99 {
        void f(int);    // does something better than the V98 version
        void f(double); // new feature
        // ...
}

// file V98.h:
namespace V98 {
        void f(int);    // does something
        // ...
}

// file Mine.h:
namespace Mine {
#include "V99.h"
#include "V98.h"
}
```

We here have a namespace **Mine** with both the latest release (**V99**) and the previous one (**V98**). If you want to be specific, you can:

```
#include "Mine.h"
using namespace Mine;
// ...
V98::f(1);      // old version
V99::f(1);      // new version
f(1);           // default version
```

The point is that the **inline** specifier makes the declarations from the nested namespace appear exactly as if they had been declared in the enclosing namespace.

This is a very ``static'' and implementer-oriented facility in that the **inline** specifier has to be placed by the designer of the namespaces -- thus making the choice for all users. It is not possible for a user of **Mine** to say ``I want the default to be **V98** rather than **V99**.''

See

- Standard 7.3.1 Namespace definition [7]-[9].

Explicit conversion operators

C++98 provides implicit and explicit constructors; that is, the conversion defined by a constructor declared **explicit** can be used only for explicit conversions whereas other constructors can be used for implicit conversions also. For example:

```
struct S { S(int); };   // "ordinary constructor" defines implicit conversion
S s1(1);                // ok
S s2 = 1;       // ok
void f(S);
f(1);           // ok (but that's often a bad surprise -- what if S was vector?)

struct E { explicit E(int); };  // explicit constructor
E e1(1);                // ok
E e2 = 1;       // error (but that's often a surprise)
void f(E);
f(1);           // error (protects against surprises -- e.g. std::vector's constructor from int is explicit)
```

However, a constructor is not the only mechanism for defining a conversion. If we can't modify a class, we can define a conversion operator from a different class. For example:

```
struct S { S(int) { } /* ... */ };

struct SS {
        int m;
        SS(int x) :m(x) { }
        operator S() { return S(m); }  // because S don't have S(SS); non-intrusive
};

SS ss(1);
S s1 = ss;      // ok; like an implicit constructor
S s2(ss);       // ok ; like an implicit constructor
void f(S);
f(ss);          // ok; like an implicit constructor
```

Unfortunately, there is no **explicit** conversion operators (because there are far fewer problematic examples). C++11 deals with that oversight by allowing conversion operators to be **explicit**. For example:

```
struct S { S(int) { } };

struct SS {
        int m;
        SS(int x) :m(x) { }
        explicit operator S() { return S(m); }  // because S don't have S(SS)
};

SS ss(1);
S s1 = ss;      // error; like an explicit constructor
S s2(ss);       // ok ; like an explicit constructor
```

```
void f(S);
f(ss);          // error; like an explicit constructor
```

See also:

- Standard: 12.3 Conversions
- [N2333=07-0193] Lois Goldthwaite, Michael Wong, and Jens Maurer: Explicit Conversion Operator (Revision 1).

Algorithms improvements

The standard library algorithms are improved partly by simple addition of new algorithms, partly by improved implementations made possible by new language features, and partly by new language features enabling easier use:

- *New algorithms*:

```
bool all_of(Iter first, Iter last, Pred pred);
bool any_of(Iter first, Iter last, Pred pred);
bool none_of(Iter first, Iter last, Pred pred);

Iter find_if_not(Iter first, Iter last, Pred pred);

OutIter copy_if(InIter first, InIter last, OutIter result, Pred pred);
OutIter copy_n(InIter first, InIter::difference_type n, OutIter result);

OutIter move(InIter first, InIter last, OutIter result);
OutIter move_backward(InIter first, InIter last, OutIter result);

pair<OutIter1, OutIter2> partition_copy(InIter first, InIter last, OutIter1 out_true, OutIter2 out_false, Pred pred);
Iter partition_point(Iter first, Iter last, Pred pred);

RAIter partial_sort_copy(InIter first, InIter last, RAIter result_first, RAIter result_last);
RAIter partial_sort_copy(InIter first, InIter last, RAIter result_first, RAIter result_last, Compare comp);
bool is_sorted(Iter first, Iter last);
bool is_sorted(Iter first, Iter last, Compare comp);
Iter is_sorted_until(Iter first, Iter last);
Iter is_sorted_until(Iter first, Iter last, Compare comp);

bool is_heap(Iter first, Iter last);
bool is_heap(Iter first, Iter last, Compare comp);
Iter is_heap_until(Iter first, Iter last);
Iter is_heap_until(Iter first, Iter last, Compare comp);

T min(initializer_list<T> t);
T min(initializer_list<T> t, Compare comp);
T max(initializer_list<T> t);
T max(initializer_list<T> t, Compare comp);
pair<const T&, const T&> minmax(const T& a, const T& b);
pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
pair<const T&, const T&> minmax(initializer_list<T> t);
pair<const T&, const T&> minmax(initializer_list<T> t, Compare comp);
pair<Iter, Iter> minmax_element(Iter first, Iter last);
pair<Iter, Iter> minmax_element(Iter first, Iter last, Compare comp);

void iota(Iter first, Iter last, T value);      // For each element referred to by the iterator i in the range [first,last), assigns *i = value and increments value as if by ++value
```

- *Effects of move*: Moving can be much more efficient than copying (see Move semantics). For example, move-based std::sort() and std::set::insert() has been measured to be 15 times faster than copy based versions. This is less impressive than it sounds because such standard library operations for standard library types, such as **string** and **vector**, are usually hand-optimized to gain the effects of moving through techniques such as replacing copies with optimized swaps. However, if *your* type has a move operation, you gain the performance benefits automatically from the standard algorithms.

  Consider also that the use of moves allows simple and efficient sort (and other algorithms) of containers of ``smart'' pointers, especially unique_ptr:

```
template<class P> struct Cmp<P> {       // compare *P values
        bool operator() (P& a, P& b) const { return *a<*b; }
}

vector<std::unique_ptr<Big>> vb;
// fill vb with unique_ptr's to Big objects

sort(vb.begin(),vb.end(),Cmp<unique_ptr<Big>>());       // don't try that with an auto_ptr
```

- *Use of lambdas*: For ages, people have complained about having to write functions or (better) function objects for use as operations, such as **Cmp<T>** above, for standard library (and other) algorithms. This was especially painful to do if you wrote large functions (don't) because in C++98 you could not define a local function object to use as an argument; now you can. However, lambdas allows us to define operations ``inline:''

```
sort(vb.begin(),vb.end(),[](unique_ptr<Big> a, unique_ptr<Big> b) { return *a<*b; });
```

  I expect lambdas to be a bit overused initially (like all powerful mechanisms).
- *Use of initializer lists*: Sometimes, initializer lists come in handy as arguments. For example, assuming **string** variables and **Nocase** being a case-insensitive comparison:

```
auto x = max({x,y,z},Nocase());
```

See also:

- 25 Algorithms library [algorithms]
- 26.7 Generalized numeric operations [numeric.ops]
- Howard E. Hinnant, Peter Dimov, and Dave Abrahams: A Proposal to Add Move Semantics Support to the C++ Language. N1377=02-0035.

Container improvements

Given the new language features and a decade's worth of experience, what has happened to the standard containers? First, of course we got a few new ones: array (a fixed-sized container), forward_list (a singly-linked list), and unordered containers (the hash tables). Next, new features, such as initializer lists, rvalue references, variadic templates, and constexpr were put to use. Consider **std::vector**.

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

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

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

```
vector<int> make_random(int n)
{
        vector<int> ref(n);
        for(auto& x : ref) x = rand_int(0,255); // some random number generator
        return ref;
}

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

The point here is that no vectors are copied. Rewrite this to return a free-store-allocated vector and you have to deal with memory management. Rewrite this to pass the vector to be filled as an argument to **make_random()** and you have a far less obvious code (plus an added opportunity for making an error).

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

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

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

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

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

An emplace takes a variadic template argument and uses that to construct an object of the desired type. Whether the **emplace_back()** really is more efficient than the **push_back()** depends on the types involved and the implementation (of the library and of variadic templates). If you think it matters, measure. Otherwise, choose based on aesthetics: **vp.push_back({s,i});** or **vp.emplace_back(s,i);**. For now, I prefer the **push_back()** version, but that might change over time.

- Scoped allocators: Containers can now hold "real allocation objects (with state)" and use those to control nested/scoped allocation (e.g. allocation of elements in a container)

Obviously, the containers are not the only parts of the standard library that has benefitted from the new language features. Consider:

- *Compile-time evaluation*: constexpr is used to ensure compiler time evaluation in , **bitset**, **duration**, **char_traits**, array, atomic types, random numbers, **complex<double>**., etc. In some cases, it means improved performance; in others (where there is no alternative to compile-time evaluation), it means absence of messy low-level code and macros.
- *Tuples*: Tuples would not be possible without variadic templates.

Scoped allocators

For compactness of container objects and for simplicity, C++98 did not require containers to support allocators with state: Allocator objects need not be stored in container objects. This is still the default in C++11, but it is possible to use an allocator with state, say an allocator that holds a pointer to an arena from which to allocate. For example:

```
template<class T> class Simple_alloc {  // C++98 style
        // no data
        // usual allocator stuff
};

class Arena {
        void* p;
        int s;
public:
        Arena(void* pp, int ss);
        // allocate from p[0..ss-1]
};

template<class T> struct My_alloc {
        Arena& a;
        My_alloc(Arena& aa) : a(aa) { }
        // usual allocator stuff
};

Arena my_arena1(new char[100000],100000);
Arena my_arena2(new char[1000000],1000000);

vector<int> v0; // allocate using default allocator

vector<int,My_alloc<int>> v1(My_alloc<int>{my_arena1}); // allocate from my_arena1

vector<int,My_alloc<int>> v2(My_alloc<int>{my_arena2}); // allocate from my_arena2

vector<int,Simple_alloc<int>> v3;       // allocate using Simple_alloc
```

Typically, the verbosity would be alleviated by the use of typedefs.

It is not guaranteed that the default allocator and **Simple_alloc** takes up no space in a vector object, but a bit of elegant template metaprogramming in the library implementation can ensure that. So, using an allocator type imposes a space overhead only if its objects actually has state (like **My_alloc**).

A rather sneaky problem can occur when using containers and user-defined allocators: Should an element be in the same allocation area as its container? For example, if you use **Your_allocator** for **Your_string** to allocate its elements and I use **My_allocator** to allocate elements of **My_vector** then which allocator should be used for string elements in **My_vector<Your_allocator>**? The solution is the ability to tell a container which allocator to pass to elements. For example, assuming that I have an allocator **My_alloc** and I want a **vector<string>** that uses **My_alloc** for both the **vector** element and **string** element allocations. First, I must make a version of **string** that accepts **My_alloc** objects:

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

Then, I must make a version of **vector** that accepts those strings, accepts a **My_alloc** object, and passes that object on to the string:

```
using svec = vector<xstring,scoped_allocator_adaptor<My_alloc<xstring>>>;
```

Finally, we can make an allocator of type **My_alloc<xstring>**:

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

Now **svec** is a **vector** of strings using **My_alloc** to allocate memory for strings. What's new is that the standard library ``adaptor'' (''wrapper type'') **scoped_allocator_adaptor** is used to indicate that string also should use **My_alloc**. Note that the adaptor can (trivially) convert **My_alloc<xstring>** to the **My_alloc<char>** that **xstring** needs.

So, we have four alternatives:

```
// vector and string use their own (the default) allocator:
using svec0 = vector<string>;
svec0 v0;

// vector (only) uses My_alloc and string uses its own (the default) allocator:
using svec1 = vector<string,My_alloc<string>>;
svec1 v1(My_alloc<string>{my_arena1});

// vector and string use My_alloc (as above):
using xstring = basic_string<char, char_traits<char>, My_alloc<char>>;
using svec2 = vector<xstring,scoped_allocator_adaptor<My_alloc<xstring>>>;
svec2 v2(scoped_allocator_adaptor<My_alloc<xstring>>{my_arena1});

// vector uses My_alloc and string uses My_string_alloc:
using xstring2 = basic_string<char, char_traits<char>, My_string_alloc<char>>;
using svec3 = vector<xstring2,scoped_allocator_adaptor<My_alloc<xstring>, My_string_alloc<char>>>;
svec3 v3(scoped_allocator_adaptor<My_alloc<xstring2>, My_string_alloc<char>>{my_arena1,my_string_arena});
```

Obviously, the first variant, **svec0**, will be by far the most common, but for systems with serious memory-related performance constraints, the other versions (especially **svec2**) can be important. A few typedefs would make that code a bit more readable, but it is good it is not something you have to write every day. The **scoped_allocator_adaptor2** is a variant of **scoped_allocator_adaptor** for the case where the two non-default allocators differ.

See also:

- Standard: 20.8.5 Scoped allocator adaptor [allocator.adaptor]
- Pablo Halpern: [The Scoped Allocator Model (Rev 2)](). N2554=08-0064.

---

std::array

The standard container **array** is a fixed-sized random-access sequence of elements defined in **<array>**. It has no space overheads beyond what it needs to hold its elements, it does not use free store, it can be initialized with an initializer list, it knows its size (number of elements), and doesn't convert to a pointer unless you explicitly ask it to. In other words, it is very much like a built-in array without the problems.

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

Note that you can have zero-length **array**s but that you cannot deduce the length of an **array** from an initializer list:

```
array<int> a3 = { 1, 2, 3 };    // error: size unknown/missing
array<int,0> a0;                // ok: no elements
int* p = a0.data();             // unspecified; don't try it
```

The standard **array**'s features makes it attractive for embedded systems programming (and similar constrained, performance-critical, or safety critical tasks). It is a sequence container so it provides the usual member types and functions (just like **vector**):

```
template<class C> C::value_type sum(const C& a)
{
        return accumulate(a.begin(),a.end(),0);
}

array<int,10> a10;
array<double,1000> a1000;
vector<int> v;
// ...
int x1 = sum(a10);
double x2 = sum(a1000);
int x3 = sum(v);
```

Also, you don't get (potentially nasty) derived to base conversions:

```
struct Apple : Fruit { /* ... */ };
struct Pear : Fruit { /* ... */ };

void nasty(array<Fruit*,10>& f)
{
        f[7] = new Pear();
};

array<Apple*,10> apples;
// ...
nasty(apples);  // error: can't convert array<Apple*,10> to array<Fruit*,10>;
```

If that was allowed, **apples** would now contain a **Pear**.

See also:

- Standard: 23.3.1 Class template array

---

std::forward_list

The standard container **forward_list**, defined in **<forward_list>**, is basically a singly-linked list. It supports forward iteration (only) and guarantees that elements don't move if you insert or erase one. It occupies minimal space (an empty list is likely to be one word) and does not provide a **size()** operation (so that it does not have to store a size member):

```
template <ValueType T, Allocator Alloc = allocator<T> >
        requires NothrowDestructible<T>
class forward_list {
public:
        // the usual container stuff
        // no size()
        // no reverse iteration
        // no back() or push_back()
};
```

See also:

- Standard: 23.3.3 Class template forward_list

Unordered containers

A unordered container is a kind of hash table. C++11 offers four standard ones:

- unordered_map
- unordered_set
- unordered_multimap
- unordered_multiset

They should have been called **hash_map** etc., but there are so many incompatible uses of those names that the committee had to choose new names and the **unordered_map**, etc. were the least bad we could find. The "unordered" refers to one of the key differences between **map** and **unordered_map**: When you iterate over a **map** you do so in the order provided by its less-than comparison operator (by default **<**) whereas the value type of **unordered_map** is not required to have a less-than comparison operator and a hash table doesn't naturally provide an order. Conversely, the element type of a **map** is not required to have a hash function.

The basic idea is simply to use **unordered_map** as an optimized version of **map** where optimization is possible and reasonable. For example:

```
map<string,int> m {
        {"Dijkstra",1972}, {"Scott",1976}, {"Wilkes",1967}, {"Hamming",1968}
};
m["Ritchie"] = 1983;
for(auto x : m) cout << '{' << x.first << ',' << x.second << '}';

unordered_map<string,int> um {
        {"Dijkstra",1972}, {"Scott",1976}, {"Wilkes",1967}, {"Hamming",1968}
};
um["Ritchie"] = 1983;
for(auto x : um) cout << '{' << x.first << ',' << x.second << '}';
```

The iterator over **m** will present the elements in alphabetical order; the iteration over **um** will not (except through a freak accident). Lookup is implemented very differently for **m** and **um**. For **m** lookup involves log2(m.size()) less-than comparisons whereas for **um** lookup involves a single call of a hash function and one or more equality operations. For a few elements (say a few dozen), it is hard to tell which is faster. For larger numbers of elements (e.g. thousands), lookup in an **unordered_map** can be much faster than for a **map**.

More to come.

See also:

- Standard: 23.5 Unordered associative containers.

---

std::tuple

The standard library **tuple** (an N-tuple) is a ordered sequence of N values where N can be a constant from 0 to a large implementation-defined value, defined in **<tuple>**. You can think of an tuple as an unnamed struct with members of the specified tuple element types. In particular, the elements of a **tuple** is stored compactly; a tuple is not a linked structure.

The element types of a tuple can explicitly specified or be deduced (using **make_tuple()**) and the elements can be access by (zero-based) index using **get()**:

```
tuple<string,int> t2("Kylling",123);

auto t = make_tuple(string("Herring"),10, 1.23);	// t will be of type tuple<string,int,double>
string s = get<0>(t);
int x = get<1>(t);
double d = get<2>(t);
```

Tuples are used (directly of indirectly) whenever we want a heterogeneous list of elements at compile time but do not want to define a named class to hold them. For example, **tuple** is used internally in std::function and std::bind to hold arguments.

The most frequently useful tuple is the 2-tuple; that is, a pair. However, pair is directly supported in the standard library through **std::pair** (20.3.3 Pairs). A **pair** can be used to initialize a **tuple**, but the opposite isn't the case.

The comparison operators (==, !=, <, <=, >, and >=) are defined for tuples of comparable element types.

See also:

- Standard: 20.5.2 Class template tuple
- Variadic template paper
- Boost::tuple

---

metaprogramming and type traits

Sorry. Come back later.

---

std::function and std::bind

The **bind** and **function** standard function objects are defined in **<functional>** (together with a lot of other function objects); they are used to handle functions and function arguments. **bind** is used to take a function (or a function object or anything you can invoke using the **(...)** syntax) and produce a function object with one or more of the arguments of the argument function ``bound'' or rearranged. For example:

```
int f(int,char,double);
auto ff = bind(f,_1,'c',1.2);	// deduce return type
int x = ff(7);			// f(7,'c',1.2);
```

This binding of arguments is usually called ``Currying.'' The **_1** is a place-holder object indicating where the first argument of **ff** is to go when **f** is called through **ff**. The first argument is called **_1**, the second **_2**, and so on. For example:

```
int f(int,char,double);
auto frev = bind(f,_3,_2,_1);	// reverse argument order
int x = frev(1.2,'c',7);	// f(7,'c',1.2);
```

Note how auto saves us from having to specify the type of the result of **bind**.

It is not possible to just bind arguments for an overloaded function, we have to explicitly state which version of an overloaded function we want to bind:

```
int g(int);
double g(double);	// g() is overloaded

auto g1 = bind(g,_1);				// error: which g()?
auto g2 = bind((double(*)(double))g,_1);	// ok (but ugly)
```

bind() comes in two variants: the one shown above and a "legacy" version where you explicitly specify the return type:

```
auto f2 = bind<int>(f,7,'c',_1);	// explicit return type
int x = f2(1.2);			// f(7,'c',1.2);
```

This second version was necessary and is widely used because the first (and for a user simplest) version cannot be implemented in C++98.

function is a type that can hold a value of just about anything you can invoke using the **(...)** syntax. In particular, the result of **bind** can be assigned to a **function**. **function** is very simple to use. For example:

```
function<float (int x, int y)> f;	// make a function object

struct int_div {			// take something you can call using ()
	float operator()(int x, int y) const { return ((float)x)/y; };
};

f = int_div();				// assign
cout << f(5, 3) << endl;		// call through the function object
std::accumulate(b,e,1,f);		// passes beautifully
```

Member functions can be treated as free functions with an extra argument

```
struct X {
	int foo(int);
};

function<int (X*, int)> f;
f = &X::foo;		// pointer to member

X x;
int v = f(&x, 5);	// call X::foo() for x with 5
function<int (int)> ff = std::bind(f,&x,_1);	// first argument for f is &x
v=ff(5);		// call x.foo(5)
```

functions are useful for callbacks, for passing operations as argument, etc. It can be seen as a replacement for the C++98 standard library function objects **mem_fun_t**, **pointer_to_unary_function**, etc. Similarly, **bind()** can be seen as a replacement for **bind1()** and **bind2()**.

See also:

- Standard: 20.7.12 Function template bind, 20.7.16.2 Class template function
- Herb Sutter: Generalized Function Pointers. August 2003.
- Douglas Gregor: Boost.Function.
- Boost::bind

---

unique_ptr

- The **unique_ptr** (defined in **<memory>**) provides a semantics of strict ownership.
  - owns the object it holds a pointer to
  - is not CopyConstructible, nor CopyAssignable; however, it is MoveConstructible and MoveAssignable.
  - stores a pointer to an object and deletes that object using the associated deleter when it is itself destroyed (such as when leaving block scope (6.7)).
- The uses of **unique_ptr** include
  - providing exception safety for dynamically allocated memory,
  - Passing ownership of dynamically allocated memory to a function
  - returning dynamically allocated memory from a function
  - storing pointers in containers
- "what **auto_ptr** should have been" (but that we couldn't write in C++98)

unique_ptr relies critically on rvalue references and move semantics.

Here is a conventional piece of exception unsafe code:

```
X* f()
{
	X* p = new X;
	// do something - maybe throw an exception
	return p;
}
```

A solution is to hold the pointer to the object on the free store in a **unique_ptr**:

```
X* f()
{
	unique_ptr<X> p(new X);		// or {new X} but not = new X
	// do something - maybe throw an exception
	return p.release();
}
```

Now, if an exception is thrown, the **unique_ptr** will (implicitly) destroy the object pointed to. That's basic RAII. However, unless we really need to return a built-in pointer, we can do even better by returning a **unique_ptr**:

```
unique_ptr<X> f()
{
	unique_ptr<X> p(new X);		// or {new X} but not = new X
	// do something - maybe throw an exception
	return p;	// the ownership is transferred out of f()
}
```

We can use this **f** like this:

```
void g()
{
	unique_ptr<X> q = f();		// move using move constructor
	q->memfct(2);			// use q
	X x = *q;			// copy the object pointed to
	// ...
}	// q and the object it owns is destroyed on exit
```

The **unique_ptr** has "move semantics" so the initialization of **q** with the rvalue that is the result of the call **f()** simply transfers ownership into **q**.

One of the uses of **unique_ptr** is as a pointer in a container, where we might have used a built-in pointer except for exception safety problems (and to guarantee destruction of the pointed to elements):

```
vector<unique_ptr<string>> vs { new string{"Doug"}, new string{"Adams"} };
```

unique_ptr is represented by a simple built-in pointer and the overhead of using one compared to a built-in pointer are miniscule. In particular, **unique_ptr** does not offer any form of dynamic checking.

See also

- the C++ draft section 20.7.10
- Howard E. Hinnant: unique_ptr Emulation for C++03 Compilers.

---

shared_ptr

A **shared_ptr** is used to represent shared ownership; that is, when two pieces of code needs access to some data but neither has exclusive ownership (in the sense of being responsible for destroying the object). A **shared_ptr** is a kind of counted pointer where the object pointed to is deleted when the use count goes to zero. Here is a highly artificial example:

```
void test()
{
	shared_ptr<int> p1(new int);	// count is 1
	{
		shared_ptr<int> p2(p1);	// count is 2
		{
			shared_ptr<int> p3(p1);	// count is 3
		}	// count goes back down to 2
	} // count goes back down to 1
}	// here the count goes to 0 and the int is deleted.
```

A more realistic example would be pointers to nodes in a general graph where someone wanting to remove a pointer to a node wouldn't know if anyone else held a pointer to that node. If a node can hold resources that require an action by a destructor (e.g. a file handle so that a file needs to be closed when the node is deleted). You could consider **shared_ptr** to be for what you might consider plugging in a garbage collector for, except that maybe you don't have enough garbage for that to be economical, your execution environment doesn't allow that, or the resource managed is not just memory (e.g. that file handle). For example:

```
struct Node {	// note: a Node may be pointed to from several other nodes.
	shared_ptr<Node> left;
	shared_ptr<Node> right;
	File_handle f;
	// ...
};
```

Here **Node**'s destructor (the implicitly generated destructor will do fine) deletes its sub-nodes; that is, **left** and **right**'s destructors are invoked. Since **left** is a **shared_ptr**, the **Node** pointed to (if any) is deleted if **left** was the last pointer to it; **right** is handled similarly and **f**'s destructor does whatever is required for **f**.

Note that you should not use a **shared_ptr** just to pass a pointer from one owner to another; that's what unique_ptr is for and **unique_ptr** does that cheaper and better. If you have been using counted pointers as return values from factory functions and the like, consider upgrading to **unique_ptr** rather than **shared_ptr**.

Please don't thoughtlessly replace pointers with **shared_ptr**s in an attempt to prevent memory leaks; **shared_ptr**s are not a panacea nor are they without costs:

- a circular linked structure of **shared_ptr**s will cause a memory leak (you'll need some logical complication to break the circle, e.g. using a **weak_ptr**),
- "shared ownership objects" tend to stay "live" for longer than scoped objects (thus causing higher average resource usage),
- shared pointers in a multi-threaded environment can be expensive (because of the need to avoid data races on the use count),
- a destructor for a shared object does not execute at a predictable time, and
- the algorithms/logic for the update of any shared object is easier to get wrong than for an object that's not shared.

. A **shared_ptr** represents *shared ownership* but shared ownership isn't my ideal: It is better if an object has a definite owner and a definite, predictable lifespan.

See also

- the C++ draft: Shared_ptr (20.7.13.3)

---

weak_ptr

Weak pointers are often explained as what you need to break loops in data structures managed using shared_ptrs. I think it is better to think of a **weak_ptr** as a pointer to something that

1. you need access to (only) if it exists, and
2. may get deleted (by someone else), and
3. must have its destructor called after its last use (usually to delete anon-memory resource)

Consider an implementation of the old "asteroid game". All asteroids are owned by "the game" but each asteroids must keep track of neighboring asteroids and handle collisions. A collision typically leads to the destruction of one or more asteroids. Each asteroid must keep a list of other asteroids in its neighborhood. Note that being on such a neighbor list should not keep an astroid "alive" (so a **shared_ptr** would be inappropriate). On the other hand, an asteroid must not be destroyed while another asteroid is looking at it (e.g. to calculate the effect of a collision). And obviously, an asteroids destructor must be called to release resources (such as a connection to the graphics system). What we need is a list of asteroids that *might* still be intact and a way of "getting hold of one *if* it exist" for a while. A **weak_ptr** does just that:

```
void owner()
{
	// ...
	vector<shared_ptr<Asteroid>> va(100);
	for (int i=0; i<va.size(); ++i) {
		// ... calculate neighbors for new asteroid ...
		va[i].reset(new Asteroid(weak_ptr<Asteroid>(va[neighbor]));
		launch(i);
	}
	// ...
}
```

reset() is the function to make a **shared_ptr** refer to a new object.

Obviously, I radically simplified "the owner" and gave each new **Asteroid** just one neighbor. The key is that we give the Asteroid a **weak_ptr** to that neighbor. The owner keeps a **shared_ptr** to represent the ownership that's shared whenever an **Asteroid** is looking (but not otherwise). The collision calculation for an **Asteroid** will look something like this:

```
void collision(weak_ptr<Asteroid> p)
{
	if (auto q = p.lock()) {	// p.lock returns a shared_ptr to p's object
		// ... that Asteroid still existed: calculate ...
	}
	else {
		// ... oops: that Asteroid has already been destroyed: just forget about it (delete the weak_ptr to it ...
	}
}
```

Note that even if the owner decides to shut down the game and deletes all **Asteroids** (by destroying the **shared_ptr**s representing ownership) every **Asteroid** that is in the middle of calculating a collision still finishes correctly (because after the **p.lock()** it holds a **shared_ptr** that won't just become invalid).

I expect to find **weak_ptr** use much rarer than "plain" **shared_ptr** use and I hope that **unique_ptr** will become much more popular than **shared_ptr** use because **unique_ptr** represents a simpler (and more efficient) notion of ownership and (therefore) allows better local reasoning.

See also

- the C++ draft: Weak_ptr (20.7.13.3)

Garbage collection ABI

Garbage collection (automatic recycling of unreferenced regions of memory) is optional in C++; that is, a garbage collector is not a compulsory part of an implementation. However, C++11 provides a definition of what a GC can do if one is used and an ABI (Application Binary Interface) to help control its actions.

The rules for pointers and lifetimes are expressed in terms of "safely derived pointer" (3.7.4.3); roughly: "pointer to something allocated by new or to a sub-object thereof." Here are some examples of "not safely derived pointers" aka "disguised pointers" aka what not to do in a program you want to be considered well behaved and comprehensible to ordinary mortals:

- Make a pointer point "elsewhere" for a while

```
int* p = new int;
p+=10;
// ... collector may run here ...
p-=10;
*p = 10;	// can we be sure that the int is still there?
```

- Hide the pointer in an int

```
int* p = new int;
int x = reinterpret_cast<int>(p);	// non-portable
p=0;
// ... collector may run here ...
p = reinterpret_cast<int*>(x);
*p = 10;	// can we be sure that the int is still there?
```

- There are many more and even nastier tricks Think I/O, think "scattering the bits around in different words", ...

There are legitimate reasons to disguise pointers (e.g. the xor trick in exceptionally memory-constrained applications), but not as many as some programmers think.

A programmer can specify where there are no pointers to be found (e.g. in an image) and what memory can't be reclaimed even if the collector can't find a pointer into it:

```
void declare_reachable(void* p);	// the region of memory starting at p
					// (and allocated by some allocator
					// operation which remembers its size)
					// must not be collected
template<class T> T* undeclare_reachable(T* p);

void declare_no_pointers(char* p, size_t n);	// p[0..n] holds no pointers
void undeclare_no_pointers(char* p, size_t n);
```

A programmer can inquire which rules for pointer safety and reclamation is in force:

```
enum class pointer_safety {relaxed, preferred, strict };
pointer_safety get_pointer_safety();
```

3.7.4.3[4]: If a pointer value that is not a safely-derived pointer value is dereferenced or deallocated, and the referenced complete object is of dynamic storage duration and has not previously been declared reachable (20.7.13.7), the behavior is undefined.

* **relaxed**: safely-derived and not safely-derived pointers are treated equivalently; like C and C++98, but that was not my intent - I wanted to allow GC if a user didn't keep a valid pointer around for an object.
* **preferred**: like relaxed; but a garbage collector may be running as a leak detector and/or detector of dereferences of "bad pointers"
* **strict**: safely-derived and not safely-derived pointers may be treated differently, i.e. a garbage collector may be running and will ignore pointers that's not safely derived

There is no standard way of saying which alternative you prefer. Considered that a "quality of implementation" and a "programming environment" issue.

See also

* the C++ draft 3.7.4.3
* the C++ draft 20.7.13.7
* Hans Boehm's GC page
* Hans Boehm's Discussion of Conservative GC
* final proposal
* Michael Spertus and Hans J. Boehm: The Status of Garbage Collection in C++0X. ACM ISMM'09.

Memory model

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

The key guarantee is: Two threads of execution can update and access separate memory locations without interfering with each other. But what is a ``memory location?'' A memory location is either an object of scalar type or a maximal sequence of adjacent bit-fields all having non-zero width. For example, here **S** has exactly four separate memory locations:

```
struct S {
        char a;                 // location #1
        int b:5,                // location #2
        int c:11,
        int :0,                 // note: :0 is "special"
        int d:8;                // location #3
        struct {int ee:8;} e;   // location #4
};
```

Why is this important? Why isn't it obvious? Wasn't this always true? The problem is that when several computations can genuinely run in parallel, that is several (apparently) unrelated instructions can execute at the same time, the quirks of the memory hardware can get exposed. In fact, in the absence of compiler support, issues of instruction and data pipelining and details of cache use *will* be exposed in ways that are completely unmanageable to the applications programmer. This is true even if no two threads have been defined to share data! Consider, two separately compiled ``threads:''

```
// thread 1:
char c;
c = 1;
int x = c;

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

For greater realism, I could have used the separate compilation (within each thread) to ensure that the compiler/optimizer wouldn't be able to eliminate memory accesses and simply ignore **c** and **b** and directly initialize **x** and **y** with 1. What are the possible values of **x** and **y**? According to C++11 the only correct answer is the obvious one: 1 and 1. The reason that's interesting is that if you take a conventional good pre-concurrency C or C++ compiler, the possible answers are 0 and 0 (unlikely), 1 and 0, 0 and 1, and 1 and 1. This has been observed ``in the wild.'' How? A linker might allocate **c** and **b** right next to each other (in the same word) -- nothing in the C or C++ 1990s standards says otherwise. In that, C++ resembles all languages not designed with real concurrent hardware in mind. However, most modern processors cannot read or write a single character, it must read or write a whole word, so the assignment to **c** really is ``read the word containing **c**, replace the **c** part, and write the word back again.'' Since the assignment to **b** is similar, there are plenty of opportunities for the two threads to clobber each other even though the threads do not (according to their source text) share data!

So, C++11 guarantees that no such problems occur for ``separate memory locations.'' More precisely: A memory location cannot be safely accessed by two threads without some form of locking unless they are both read accesses. Note that different bitfields within a single word are not separate memory locations, so don't share structs with bitfields among threads without some form of locking. Apart from that caveat, the C++ memory model is simply ``as everyone would expect.''

However, it is not always easy to think straight about low-level concurrency issues. Consider:

```
// start with x==0 and y==0

if (x) y = 1;   // Thread 1

if (y) x = 1;   // Thread 2
```

Is there a problem here? More precisely, is there a data race? (No there isn't).

Fortunately, we have already adapted to modern times and every current C++ compiler (that I know of) gives the one right answer and have done so for years. They do so for most (but unfortunately not yet for all) tricky questions. After all, C++ has been used for serious systems programming of concurrent systems ``forever.'' The standard should further improve things.

See also

* Standard: 1.7 The C++ memory model [intro.memory]
* Paul E. McKenney, Hans-J. Boehm, and Lawrence Crowl: C++ Data-Dependency Ordering: Atomics and Memory Model. N2556==08-0066.
* Hans-J. Boehm: Threads basics, HPL technical report 2009-259. ``what every programmer should know about memory model issues.''
* Hans-J. Boehm and Paul McKenney: A slightly dated FAQ on C++ memory model issues.

Threads

A thread is a representation of an execution/computation in a program. In C++11, as in much modern computing, a thread can -- and usually does -- share an address space with other threads. In this, it differs from a process, which generally does not directly share data with other processes. C++ have had a host of threads implementations for a variety of hardware and operating systems in the past, what's new is a standard-library threads library, a standard thread ABI.

Many thick books and tens of thousands of papers have been writing about concurrency, parallelism, and threading, this FAQ entry barely scratch the surface. It is *hard* to think clearly about concurrency. If you want to do concurrent programming, at least read a book. Do not rely just on a manual, a standard, or an FAQ.

A thread is launched by constructing a std::thread with a function or a function object (incl. a lambda):

```
#include<thread>

void f();

struct F {
        void operator()();
};

int main()
{
        std::thread t1{f};      // f() executes in separate thread
        std::thread t2{F()};    // F()() executes in separate thread
}
```

Unfortunately, this is unlikely to give any useful results -- whatever **f()** and **F()** might do. The snag is that the program may terminate before or after **t1** executes **f()** and before or after **t2** executes **F()**. We need to wait for the two tasks to complete:

```
int main()
{
	std::thread t1{f};	// f() executes in separate thread
	std::thread t2{F()};	// F()() executes in separate thread

	t1.join();	// wait for t1
	t2.join();	// wait for t2
}
```

The **join()**s ensure that we don't terminate until the threads have completed. To ``join'' means to ``wait for the thread to terminate.''

Typically, we'd like to pass some arguments to the task to be executed (I call something executed by a thread a task). For example:

```
void f(vector<double>&);

struct F {
	vector<double>& v;
	F(vector<double>& vv) :v{vv} { }
	void operator()();
};

int main()
{
	std::thread t1{std::bind(f,some_vec)};	// f(some_vec) executes in separate thread
	std::thread t2{F(some_vec)};		// F(some_vec)() executes in separate thread

	t1.join();
	t2.join();
}
```

Basically, the standard library bind makes a function object of its arguments.

In general, we'd also like to get a result back from an executed task. With plain tasks, there is no notion of a return value; I recommend std::future for that. Alternative, we can pass an argument to a task telling it where to put its result: For example:

```
void f(vector<double>&, double* res);	// place result in res

struct F {
	vector<double>& v;
	double* res;
	F(vector<double>& vv, double* p) :v{vv}, res{p} { }
	void operator()();	// place result in res
};

int main()
{
	double res1;
	double res2;

	std::thread t1{std::bind(f,some_vec,&res1)};	// f(some_vec,&res1) executes in separate thread
	std::thread t2{F(some_vec,&res2)};		// F(some_vec,&res2)() executes in separate thread

	t1.join();
	t2.join();

	std::cout << res1 << ' ' << res2 << '\n';
}
```

But what about errors? What if a task throws an exception? If a task throws an exception and doesn't catch it itself **std::terminate()** is called. That typically means that the program finishes. We usually try rather hard to avoid that. A std::future can transmit an exception to the parent/calling thread; that's one reason I like futures. Otherwise, return some sort of error code.

When a **thread** goes out of scope the program is **terminate()**d unless its task has completed. That's obviously to be avoided.

There is no way to request a **thread** to terminate (i.e. request that it exit as a soon as possible and as gracefully as possible) or to force a thread to terminate (i.e. kill it). We are left with the options of

- designing our own cooperative ``interruption mechanism'' (with a piece of shared data that a caller thread can set for a called thread to check (and quickly and gracefully exit when it is set)),
- ``going native'' (using **thread::native_handle()** to gain access to the operating system's notion of a thread),
- kill the process (**std::quick_exit()**),
- kill the program (**std::terminate()**).

This was all the committee could agree upon. In particular, representatives from POSIX were vehemently against any form of ``thread cancellation'' however much C++'s model of resources rely on destructors. There is no perfect solution for every systems and every possible application.

The basic problem with threads is data races; that is, two threads running in a single address space can independently access an object in ways that cause undefined results. If one (or both) writes to the object and the other (or both) reads the object they have a ``race'' for who gets its operation(s) done first. The results are not just undefined; they are usually completely unpredictable. Consequently, C++11 provides some rules/guarantees for the programmer to avoid data races:

- A C++ standard library function shall not directly or indirectly access objects accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's arguments, including **this**.
- A C++ standard library function shall not directly or indirectly modify objects accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's nonconst arguments, including **this**.
- C++ standard library implementations are required to avoid data races when different elements in the same sequence are modified concurrently.

Concurrent access to a stream object, stream buffer object, or C Library stream by multiple threads may result in a data race unless otherwise specified. So don't share an output stream between two threads unless you somehow control the access to it.

You can

- wait for a thread for a specified time
- control access to some data by mutual exclusion
- control access to some data using locks
- wait for an action of another task using a condition variable
- return a value from a thread through a future

See also

- Standard: 30 Thread support library [thread]
- 17.6.4.7 Data race avoidance [res.on.data.races]
- ???
- H. Hinnant, L. Crowl, B. Dawes, A. Williams, J. Garland, et al.: Multi-threading Library for Standard C++ (Revision 1) N2497==08-0007
- H.-J. Boehm, L. Crowl: C++ object lifetime interactions with the threads API N2880==09-0070.
- L. Crowl, P. Plauger, N. Stoughton: Thread Unsafe Standard Functions N2864==09-0054.
- WG14: Thread Cancellation N2455=070325.

Mutual exclusion

A **mutex** is a primitive object use for controlling access in a multi-threaded system. The most basic use is

```
std::mutex m;
int sh; // shared data
// ...
m.lock();
// manipulate shared data:
sh+=1;
m.unlock();
```

Only one thread at a time can be in the region of code between the **lock()** and the **unlock()** (often called a critical region). If a second thread tries **m.lock()** while a first thread is executing in that region, that second thread is blocked until the first executes the **m.unlock()**. This is simple. What is not simple is to use mutexes in a way that doesn't cause serious problems: What if a thread ``forgets'' to **unlock()**? What if a thread tries to **lock()** the same mutex twice? What if a thread waits a very long time before doing an **unlock()**? What if a thread need to **lock()** two mutexes to do its job? The complete answers fill books. Here (and in the Locks section) are just the raw basics.

In addition to **lock()**, a **mutex** has a **try_lock()** operation which can be used to try to get into the critical region without the risk of getting blocked:

```
std::mutex m;
int sh; // shared data
// ...
if (m.try_lock()) {
        // manipulate shared data:
        sh+=1;
        m.unlock();
else {
        // maybe do something else
}
```

A **recursive_mutex** is a **mutex** that can be acquired more than once by a thread:

```
std::recursive_mutex m;
int sh; // shared data
// ...
void f(int i)
{
        // ...
        m.lock();
        // manipulate shared data:
        sh+=1;
        if (--i>0) f(i);
        m.unlock();
        // ...
}
```

Here, I have been blatant and let **f()** call itself. Typically, the code is more subtle. The recursive call will be indirect along the line of **f()** calls **g()** that calls **h()** that calls **f()**.

What if I need to acquire a **mutex** within the next ten seconds? The **timed_mutex** class is offered for that. Its operations are specialized versions of **try_lock()** with an associated time limit:

```
std::timed_mutex m;
int sh; // shared data
// ...
if (m.try_lock_for(std::chrono::seconds(10))) {
        // manipulate shared data:
        sh+=1;
        m.unlock();
}
else {
        // we didn't get the mutex; do something else
}
```

The **try_lock_for()** takes a relative time, a duration as its argument. If instead you want to wait until a fixed point in time, a time_point you can use **try_lock_until()**:

```
std::timed_mutex m;
int sh; // shared data
// ...
if (m.try_lock_until(midnight)) {
        // manipulate shared data:
        sh+=1;
        m.unlock();
}
else {
        // we didn't get the mutex; do something else
}
```

The **midnight** is a feeble joke: for a mechanism as low level as mutexes, the timescale is more likely to be milliseconds than hours.

There is of course also a **recursive_timed_mutex**.

A mutex is considered a resource (as it is typically used to represent a real resource) and must be visible to at least two threads to be useful. Consequently, it cannot be copied or moved (you couldn't just make another copy of a hardware input register).

It can be surprisingly difficult to get the **lock()**s and **unlock()**s to match. Think of complicated control structures, errors, and exceptions. If you have a choice, use locks to manage your mutexes; that will save you and your users a lot of sleep.

See also

- Standard: 30.4 Mutual exclusion [thread.mutex]
- H. Hinnant, L. Crowl, B. Dawes, A. Williams, J. Garland, et al.: Multi-threading Library for Standard C++ (Revision 1)
- ???

Locks

A **lock** is an object that can hold a reference to a **mutex** and may **unlock()** the **mutex** during the **lock**'s destruction (such as when leaving block scope). A **thread** may use a **lock** to aid in managing **mutex** ownership in an exception safe manner. In other words, a lock implements Resource Acquisition Is Initialization for mutual exclusion. For example:

```
std::mutex m;
int sh; // shared data
// ...
void f()
{
        // ...
        std::unique_lock lck(m);
        // manipulate shared data:
        sh+=1;
}
```

A lock can be moved (the purpose of a lock is to represent local ownership of a non-local resource), but not copied (which copy would own the resource/**mutex**?).

This straightforward picture of a lock is clouded by **unique_lock** having facilities to do just about everything a mutex can, but safer. For example, we can use a lock to do try lock:

```
std::mutex m;
int sh; // shared data
```

```
// ...
void f()
{
        // ...
        std::unique_lock lck(m,std::defer_lock);        // make a lock, but don't acquire the mutex
        // ...
        if (lck.try_lock()) {
                // manipulate shared data:
                sh+=1;
        }
        else {
                // maybe do something else
        }
}
```

Similarly, **unique_lock** supports **try_lock_for()** and try_lock_until(). What you get from using a lock rather than the mutex directly is exception handling and protection against forgetting to **unlock()**. In concurrent programming, we need all the help we can get.

What if we need two resources represented by two mutexes? The naive way is to acquire the mutexes in order:

```
std::mutex m1;
std::mutex m2;
int sh1;        // shared data
int sh2
// ...
void f()
{
        // ...
        std::unique_lock lck1(m1);
        std::unique_lock lck2(m2);
        // manipulate shared data:
        sh1+=sh2;
}
```

This has the potentially deadly flaw that some other thread could try to acquire **m1** and **m2** in the opposite order so that each had one of the locks needed to proceed and would wait forever for the second (that's called deadlock). With many locks in a system, that's a real danger. Consequently, the standard locks provide two functions for (safely) trying to acquire two or more locks:

```
void f()
{
        // ...
        std::unique_lock lck1(m1,std::defer_lock);      // make locks but don't yet try to acquire the mutexes
        std::unique_lock lck2(m2,std::defer_lock);
        std::unique_lock lck3(m3,std::defer_lock);
        lock(lck1,lck2,lck3);
        // manipulate shared data
}
```

Obviously, the implementation of **lock()** has to be carefully crafted to avoid deadlock. In essence, it will do the equivalent to careful use of **try_lock()**s. If **lock()** fails to acquire all locks it will throw an exception. Actually, **lock()** can take any argument with **lock()**, **try_lock()**, and **unlock()** member functions (e.g. a **mutex**), so we can't be specific about which exception **lock()** might throw; that depends on its arguments.

If you prefer to use **try_lock()**s yourself, there is an equivalent to **lock()** to help:

```
void f()
{
        // ...
        std::unique_lock lck1(m1,std::defer_lock);      // make locks but don't yet try to acquire the mutexes
        std::unique_lock lck2(m2,std::defer_lock);
        std::unique_lock lck3(m3,std::defer_lock);
        int x;
        if ((x = try_lock(lck1,lck2,lck3))==-1) {       // welcome to C land
                // manipulate shared data
        }
        else {
                // x holds the index of a mutex we could not acquire
                // e.g. if lck2.try_lock() failed x==1
        }
}
```

See also

- Standard: 30.4.3 Locks [thread.lock]
- ???

Condition variables

Condition variables provide synchronization primitives used to block a thread until notified by some other thread that some condition is met or until a system time is reached.

Sorry, I have not had time to write this entry. Please come back later.

See also

- Standard: 30.5 Condition variables [thread.condition]
- ???

Time utilities

We often want to time things or to do things dependent on timing. For example, the standard-library mutexes and locks provide the option for a thread to wait for a period of time (a **duration**) or to wait until a given point in time (a **time_point**).

If you want to know the current **time_point** you can call **now()** for one of three **clock**s: **system_clock**, **monotonic_clock**, **high_resolution_clock**. For example:

```
monotonic_clock::time_point t = monotonic_clock::now();
// do something
monotonic_clock::duration d = monotonic_clock::now() - t;
// something took d time units
```

A **clock** returns a **time_point**, and a **duration** is the difference of two **time_point**s from the same **clock**. As usual, if you are not interested in details, **auto** is your friend:

```
auto t = monotonic_clock::now();
// do something
auto d = monotonic_clock::now() - t;
// something took d time units
```

The time facilities here are intended to efficiently support uses deep in the system; they do not provide convenience facilities to help you maintain your social calendar. In fact, the time facilities originated with the stringent needs for high-energy physics. To be able to express all time scales (such as centuries and picoseconds), avoid confusion about units, typos, and rounding errors, **duration**s and **time_point**s are expressed using a compile-time rational number package. A **duration** has two parts: a numbers clock ``tick'' and something (a ``period'') that says what a tick means (is it a second or a millisecond?); the period is part of a **durations** type. This table from the standard header **<ratio>**, defining the periods of the SI system (also known as MKS or metric system) might give you an idea of the scope of use:

```
// convenience SI typedefs:
```

```
typedef ratio<1, 1000000000000000000000000> yocto;  // conditionally supported
typedef ratio<1,    1000000000000000000000> zepto;  // conditionally supported
typedef ratio<1,       1000000000000000000> atto;
typedef ratio<1,          1000000000000000> femto;
typedef ratio<1,             1000000000000> pico;
typedef ratio<1,                1000000000> nano;
typedef ratio<1,                   1000000> micro;
typedef ratio<1,                      1000> milli;
typedef ratio<1,                       100> centi;
typedef ratio<1,                        10> deci;
typedef ratio<                       10, 1> deca;
typedef ratio<                      100, 1> hecto;
typedef ratio<                     1000, 1> kilo;
typedef ratio<                  1000000, 1> mega;
typedef ratio<               1000000000, 1> giga;
typedef ratio<            1000000000000, 1> tera;
typedef ratio<         1000000000000000, 1> peta;
typedef ratio<      1000000000000000000, 1> exa;
typedef ratio<   1000000000000000000000, 1> zetta;  // conditionally supported
typedef ratio<1000000000000000000000000, 1> yotta;  // conditionally supported
```

The compile time rational numbers provide the usual arithmetic (**+**, **-**, *****, and **/**) and comparison (==, !=, <, <=, >, >=) operators for whatever combinations **duration**s and **time_point**s makes sense (e.g. you can't add two time_points). These operations are also checked for overflow and divide by zero. Since this is a compile-time facility, don't worry about run-time performance. In addition you can use ++, --, +=, -=, *=, and /= on **duration**s and **tp+=d** and **tp-=d** for a **time_point tp** and a **duration d**.

Here are some examples of values using standard **duration** types as defined in **<chrono>**:

```
microseconds mms = 12345;
milliseconds ms = 123;
seconds s = 10;
minutes m = 30;
hours h = 34;

auto x = std::chrono::hours(3);                // being explicit about namespaces
auto x = hours(2)+minutes(35)+seconds(9);      // assuming suitable "using"
```

You cannot initialize a **duration** to a fraction. For example, don't try 2.5 seconds; instead use 2500 milliseconds. This is because a because a **duration** is interpreted as a number of ``ticks.'' Each tick represent on unit of the **duration**'s ``period,'' such as **milli** and **kilo** as defined above. The default unit is **seconds**; that is, for a **duration** with a period of 1 a tick is interpreted as a second. We can be explicit about the representation of a **duration**:

```
duration<long> d0 = 5;                 // seconds (by default)
duration<long,kilo> d1 = 99;           // kiloseconds!
duration<long,ratio<1000,1>> d2 = 100;  // d1 and d2 have the same type ("kilo" means "*1000")
```

If we actually want to do something with a **duration**, such as writing it out, we have to give a unit, such as **minutes** or **microseconds**. For example:

```
auto t = monotonic_clock::now();
// do something
nanoseconds d = monotonic_clock::now() - t;     // we want the result in nanoseconds
cout << "something took " << d << "nanoseconds\n";
```

Alternatively, we could convert the **duration** to a floating point number (to get rounding):

```
auto t = monotonic_clock::now();
// do something
auto d = monotonic_clock::now() - t;
cout << "something took " << duration_cast<double>(d).count() << "seconds\n";
```

The **count()** is the number of ``ticks.''

See also

- Standard: 20.9 Time utilities [time]
- Howard E. Hinnant, Walter E. Brown, Jeff Garland, and Marc Paterno: A Foundation to Sleep On. N2661=08-0171. Including ``A Brief History of Time'' (With apologies to Stephen Hawking).

Atomics

Sorry, I have not had time to write this entry. Please come back later.

See also

- Standard: 29 Atomic operations library [atomics]
- ???

---

std::future and std::promise

Concurrent programming can be *hard*, especially if you try to be clever with threads, and locks. It is harder still if you must use condition variables or use std-atomics (for lock-free programming). C++11 offers **future** and **promise** for returning a value from a task spawned on a separate thread and **packaged_task** to help launch tasks. The important point about **future** and **promise** is that they enable a transfer of a value between two tasks without explicit use of a lock; ``the system'' implements the transfer efficiently. The basic idea is simple: When a task wants to return a value to the **thread** that launched it, it puts the value into a **promise**. Somehow, the implementation makes that value appear in the **future** attached to the **promise**. The caller (typically the launcher of the task) can then read the value. For added simplicity, see async().

The standard provides three kinds of futures, **future** for most simple uses, and **shared_future** and **atomic_future** for some trickier cases. Here, I'll just present **future** because it's the simplest and does all I need done. If we have a **future<X>** called **f**, we can **get()** a value of type **X** from it:

```
X v = f.get();  // if necessary wait for the value to get computed
```

If the value isn't there yet, our thread is blocked until it arrives. If the value couldn't be computed, the result of **get()** might be to throw an exception (from the system or transmitted from the task from which we were trying to **get()** the value.

We might not want to wait for a result, so we can ask the **future** if a result has arrived:

```
if (f.wait_for(0)) {    // there is a value to get()
        // do something
}
else {
        // do something else
}
```

However, the main purpose of **future** is to provide that simple **get()**.

The main purpose of **promise** is to provide a simple ``put'' (curiously, called ``set'') to match **future**'s **get()**. The names ``future'' and ``promise'' are historical; please don't blame me. They are also a fertile source of puns.

If you have a **promise** and need to send a result of type **X** (back) to a **future**, there are basically two things you can do: pass a value and pass an exception:

```
try {
        X res;
        // compute a value for res
        p.set_value(res);
}
```

```
catch (...) {   // oops: couldn't compute res
        p.set_exception(std::current_exception());
}
```

So far so good, but how do I get a matching **future/promise** pair, one in my **thread** and one in some other **thread**? Well, since **future**s and **promise**s can be moved (not copied) around there is a wide variety of possibilities. The most obvious idea is for whoever wants a task done to create a thread and give the promise to it while keeping the corresponding future as the place for the result. Using async() is the most extreme/elegant variant of the latter technique.

The **packaged_task** type is provided to simplify launching a thread to execute a task. In particular, it takes care of setting up a **future** connected to a **promise** and to provides the wrapper code to put the return value or exception from the task into the **promise**. For example:

```
double comp(vector<double>& v)
{
        // package the tasks:
        // (the task here is the standard accumulate() for an array of doubles):
        packaged_task<double(double*,double*,double)> pt0{std::accumulate<double*,double*,double>};
        packaged_task<double(double*,double*,double)> pt1{std::accumulate<double*,double*,double>};

        auto f0 = pt0.get_future();     // get hold of the futures
        auto f1 = pt1.get_future();

        pt0(&v[0],&v[v.size()/2],0);    // start the threads
        pt1(&v[v.size()/2],&v[size()],0);

        return f0.get()+f1.get();       // get the results
}
```

See also

- Standard: 30.6 Futures [futures]
- Anthony Williams: Moving Futures - Proposed Wording for UK comments 335, 336, 337 and 338. N2888==09-0078.
- Detlef Vollmann, Howard Hinnant, and Anthony Williams An Asynchronous Future Value (revised) N2627=08-0137.
- Howard E. Hinnant: Multithreading API for C++0X - A Layered Approach. N2094=06-0164. The original proposal for a complete threading package..

---

std::async()

The **async()** simple task launcher function is the only facility in this FAQ that has not yet been voted into the draft standard. I expect it to be voted in in October after reconciling two slighty different proposals (feel free to tell your local committee member to be sure to vote for it).

Here is an example of a way for the programmer to rise above the messy threads-plus-lock level of concurrent programming:

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

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

        auto f0 {async(Accum{&v[0],&v[v.size()/4],0.0})};
        auto f1 {async(Accum{&v[v.size()/4],&v[v.size()/2],0.0})};
        auto f2 {async(Accum{&v[v.size()/2],&v[v.size()*3/4],0.0})};
        auto f3 {async(Accum{&v[v.size()*3/4],&v[v.size()],0.0})};

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

This is a very simple-minded use of concurrency (note the ``magic number''), but note the absence of explicit **thread**s, **lock**s, buffers, etc. The type of the **f**-variables are determined by the return type of the standard-library function **async()** which is a future. If necessary, **get()** on a **future** waits for a thread to finish. Here, it is **async()**'s job to spawn **thread**s as needed and the **future**'s job to **join()** the **thread**s appropriately. ``Simple'' is the most important aspect of the **async()/future** design; futures can also be used with threads in general, but don't even *think* of using **async()** to launch tasks that do I/O, manipulates mutexes, or in other ways interact with other tasks. The idea behind **async()** is the same as the idea behind the range-**for** statement: Provide a simple way to handle the simplest, rather common, case and leave the more complex examples to the fully general mechanism.

An **async()** can be requested to launch in a new **thread**, in any **thread** but the caller's, or to launch in a different **thread** only if **async()** ``thinks'' that it is a good idea. The latter is the simplest from the user's perspective and potentially the most efficient (for *simple* tasks(only)).

See also

- Standard: ???
- Lawrence Crowl: An Asynchronous Call for C++. N2889 = 09-0079.
- Herb Sutter : A simple async() N2901 = 09-0091 .

---

abandoning a process

Sorry, I have not had time to write this entry. Please come back later.

- Abandoning a process

Random number generation

Random numbers are useful in many contexts, such as testing, games, simulation, and security. The diversity of application areas is reflected in the wide selection of random number generators provided by the standard library. A random number generator consists of two parts an *engine* that produces a sequence of random or pseudo-random values and a *distribution* that maps those values in to a mathematical distribution in a range. Examples of distributions are **uniform_int_distribution** (where all integers produced are equally likely) and **normal_distribution** (``the bell curve''); each for some specified range. For example:

```
uniform_int_distribution<int> one_to_six {1,6};  // distribution that maps to the ints 1..6
default_random_engine re {};                     // the default engine
```

To get a random number, you call a distribution with an engine:

```
int x = one_to_six(re); // x becomes a value in [1:6]
```

Passing the engine in every call can be considered tedious, so we could bind that argument to get a function object that we can call without arguments:

```
auto dice {bind(one_to_six,re)};   // make a generator

int x = dice(); // roll the dice: x becomes a value in [1:6]
```

Thanks to its uncompromising attention to generality and performance one expert deemed the standard-library random number component ``what every random number library wants to be when it grows up.'' However, it can hardly be deemed ``novice friendly.'' I have never found the random number interface to be a performance bottle neck, but I have never taught novices (of any background) without needing a very simple random number generator. This would be sufficient

```
int rand_int(int low, int high);        // generate a random number from a uniform distribution in [low:high]
```

So, how could we get that? We have to get something like **dice()** inside **rand_int()**:

```
int rand_int(int low, int high)
{
	static default_random_engine re {};
	using Dist = uniform_int_distribution<int>;
	static Dist uid {};
	return uid(re, Dist::param_type{low,high});
}
```

That definition is still ``expert level'' but the *use* of **rand_int()** manageable in the first week of a C++ course.

Just to show a non-trivial example, here is a program that generates and prints a normal distribution:

```
default_random_engine re;   // the default engine
normal_distribution<int> nd(31 /* mean */,8 /* sigma */);

auto norm = std::bind(nd, re);

vector<int> mn(64);

int main()
{
	for (int i = 0; i<1200; ++i) ++mn[round(norm())]; // generate

	for (int i = 0; i<mn.size(); ++i) {
		cout << i << '\t';
		for (int j=0; j<mn[i]; ++j) cout << '*';
		cout << '\n';
	}
}
```

The result was:

```
0	
1	
2	
3	
4	*
5	
6	
7	
8	
9	*
10	***
11	***
12	***
13	*****
14	*******
15	****
16	**********
17	***********
18	****************
19	*******************
20	*******************
21	**************************
22	**********************************
23	**********************************************
24	********************************************
25	*****************************************
26	*********************************************
27	*********************************************************
28	***************************************************
29	******************************************************************
30	**********************************************
31	*********************************************************************
32	**********************************************
33	*************************************************************
34	**************************************************************
35	***************************************
36	***********************************************
37	**********************************************
38	*********************************************
39	********************************
40	********************************************
41	***********************
42	**************************
43	******************************
44	*****************
45	*************
46	*********
47	********
48	*****
49	*****
50	****
51	***
52	***
53	**
54	*
55	*
56	
57	*
58	
59	
60	
61	
62	
63	
```

See also

- Standard 26.5: Random number generation

Regular expressions

- 28 Regular expressions library

Sorry, I have not had time to write this entry. Please come back later.

See also

- Standard: ???
- ???

Concepts

Warning: "concepts" did not make it into C++11 and a radical redesign is in progress

"Concepts" is a mechanism for describing requirements on types, combinations of types, and combinations of types and integers. It is particularly useful for getting early checking of uses of templates. Conversely, it also helps early detection of errors in a template body. Consider the standard library algorithm **fill**:

```
template<ForwardIterator Iter, class V>          // types of types
        requires Assignable<Iter::value_type,V>  // relationships among argument types
void fill(Iter first, Iter last, const V& v);       // just a declaration, not definition


fill(0, 9, 9.9);                // Iter is int; error: int is not a ForwardIterator
                                //                      int does not have a prefix *
fill(&v[0], &v[9], 9.9);        // Iter is int*; ok: int* is a ForwardIterator
```

Note that we only declared **fill()**; we did not define it (provide its implementation). On the other hand, we explicitly stated what **fill()** requires from its argument:

- The arguments **first** and **last** must be of a type that is a **ForwardIterator** (and they must be of the same type).
- The third argument **v** must be of a type that can be assigned to the **ForwardIterator**'s **value_type**.

We knew that, of course, having read the standard. However, compilers do not read requirement documents, so we had to tell it in code using the concepts **ForwardIterator** and **Assignable**. The result is that errors in the use of **fill()** are caught immediately at the point of use and that error messages are greatly improved. The compiler now has the information about the programmers' intents to allow good checking and good diagnostics.

Concepts also help template implementers. Consider:

```
template<ForwardIterator Iter, class V>
        requires Assignable<Iter::value_type,V>
void fill(Iter first, Iter last, const V& v)
{
        while (first!=last) {
                *first = v;
                first=first+1;  // error: + not defined for ForwardIterator
                                // (use ++first)
        }
}
```

This error is caught immediately, eliminating the need for much tedious testing (though of course not all testing).

Being able to classify and distinguish different types of types, we can overload based on the kind of types passed. For example

```
// iterator-based standard sort (with concepts):
template<Random_access_iterator Iter>
        requires Comparable<Iter::value_type>
void sort(Iter first, Iter last); // use the usual implementation

// container-based sort:
template<Container Cont>
        requires Comparable<Cont::value_type>
void sort(Cont& c)
{
        sort(c.begin(),c.end());    // simply call the iterator version
}

void f(vector<int>& v)
{
        sort(v.begin(), v.end());   // one way
        sort(v);                    // another way
        // ...
}
```

You can define your own concepts, but for starters the standard library provides a variety of useful concepts, such as **ForwardIterator**, **Callable**, **LessThanComparable**, and **Regular**.

Note: the C++0x standard libraries were specified using concepts.

See also

- the C++ draft 14.10 Concepts
- [N2617=08-0127] Douglas Gregor, Bjarne Stroustrup, James Widman, and Jeremy Siek: Proposed Wording for Concepts (Revision 5) (Final proposal).
- Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine: Concepts: Linguistic Support for Generic Programming in C++. OOPSLA'06, October 2006.

Concept maps

An **int*** is a **ForwardIterator**; we said so when presenting concepts, the standard has always said so, and even the first version of the STL used pointers as iterators. However, we also talked about **ForwardIterator**'s **value_type**. But an **int*** does not have a member called **value_type**; in fact, it has no members. So how can an **int*** be a **ForwardIterator**? It is because we say it is. Using a **concept_map**, we say that when a **T*** is used where a **ForwardIterator** is required, we consider the **T** its **value_type**:

```
template<Value_type T>
concept_map ForwardIterator<T*> {         // T*'s value_type is T
        typedef T value_type;
};
```

A **concept_map** allows us to say how we want to see a type, saving us from having to modify it or to wrap it into a new a type. "Concept maps" is a very flexible and general mechanism for adapting independently developed software for common use.

- the C++ draft 14.10.2 Concept maps
- [N2617=08-0127] Douglas Gregor, Bjarne Stroustrup, James Widman, and Jeremy Siek: Proposed Wording for Concepts (Revision 5) (Final proposal).
- Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, Andrew Lumsdaine: Concepts: Linguistic Support for Generic Programming in C++. OOPSLA'06, October 2006.

Axioms

An axiom is a set of predicates specifying the semantics of a concept. The primary use cases for axioms are external tools (e.g. not the common compiler actions), such as tools for domain-specific optimizations (languages for specifying program transformations were a significant part of the motivation for axioms). A secondary use is simply precise specification of semantics in the standard (as is used in many parts of the standard library specification). Axioms may also be useful for some optimizations (done by compilers and traditional optimizers), but compilers are *not* required to take notice of user-supplied axioms; they work based on the semantics defined by the standard.

An axiom lists pairs of computations that may be considered equivalent. Consider:

```
concept Semigroup<typename Op, typename T> : CopyConstructible<T> {
        T operator()(Op, T, T);
        axiom Associativity(Op op, T x, T y, T z) {
                op(x, op(y, z)) <=> op(op(x, y), z);    // T's operator may be assumed to be associative
        }
}

concept Monoid<typename Op, typename T> : Semigroup<Op, T> {    // a monoid is a semigroup with an identity element
        T identity_element(Op);
        axiom Identity(Op op, T x) {
                op(x, identity_element(op)) <=> x;
                op(identity_element(op), x) <=> x;
        }
}
```

The **<=>** is the equivalence operator, which is used only in axioms. Note that you cannot (in general) prove an axiom; we use axioms to state what we cannot prove, but what a programmer can state to be an acceptable assumption. Note that both sides of an equivalence statement may be illegal for some values, e.g. use of a NaN (not a number) for a floating-point type: if both sides of an equivalence uses a NaN both are (obviously) invalid and equivalent (independently of what the axiom says), but if only one side uses a NaN there may be opportunities for taking advantage of the axiom.

An axiom is a sequence of equivalence statements (using **<=>**) and conditional statements (of the form "if (something) then we may assume the following equivalence"):

```
// in concept TotalOrder:
axiom Transitivity(Op op, T x, T y, T z)
{
        if (op(x, y) && op(y, z)) op(x, z) <=> true;    // conditional equivalence
}
```

See also

- the C++ draft 14.9.1.4 Axioms
- ???.