Improving Standard C++ for the Physics Community

M. Paterno*, W. E. Brown†, FNAL, Batavia, IL 60510, USA

Abstract

As Fermilab's representatives to the C++ standardization effort, we have been promoting directions of special interest to the physics community. We here report on selected recent developments toward the next revision, informally denoted *C++0x* , of the C++ Standard.

INTRODUCTION

Standard C++ [1, 2] has become the *lingua franca* of high energy physics (HEP) computing. Among the many reasons for this are the language's great expressive power, its high computational efficiency, and its support for multiple programming paradigms. In addition, high-quality C++ compilers, including free compilers, are available in nearly all environments.

Historically, the HEP community, and the scientific community in general, has had little involvement with the C++ standardization process. The authors believed this situation could be improved, and in 2000, with encouragement from C++ originator Bjarne Stroustrup, obtained support from the Fermilab Computing Division to participate in the C++ standardization process.

C++ Standards Bodies and Their Work

There are actually *multiple* C++ standards committees. The international committee (designated ISO JTC1-SC22/WG21) is one; its members are national standards bodies.[1] Fermilab is a voting member of the US national committee, ANSI NCITS/J16. WG21 and J16 conduct co-located meetings on a regular basis, and share a common web site [3]. While maintaining independent formal structures (such as voting procedures), the two bodies' technical experts nonetheless work together sufficiently closely that they typically refer to the joint body as simply *the Committee*, only rarely needing to distinguish among the organizations. We will continue their usage in this paper.

ISO published the first C++ standard [1] in 1998. For two years thereafter the Committee enforced what Stroustrup referred to as "a period of calm to enhance the stability of the language." The Committee used this time to identify, evaluate, and consolidate numerous minor editorial adjustments and technical clarifications to the language of the original standard. The resulting updated C++ standard, known as *C++03* , was published by ISO in 2003 [2] and is also available as [4].

In parallel with the ISO publication cycle, the Committee began soliciting and evaluating proposals for extensions to the C++ language and to its library. A Technical Report on the standard library is likely to be voted out of committee in October 2004. To date, the Committee continues to solicit and evaluate proposals.

Goals and Guidelines

In order to manage the future development of a project, it is necessary to have a vision of the intended future. Loosely stating such a vision for C++, the Committee intends:

- to minimize incompatibilities with C++03 and with C99 [5],
- to keep to the *zero-overhead* principle,
- to maintain or increase type safety, and
- to minimize *implementation-defined* and *undefined* behaviors.

Minimizing incompatibilities with C++03 preserves users' investment in existing C++ code. Minimizing incompatibilities with C99 provides similar benefit to user communities that program in mixed-language environments.

The zero-overhead principle, "What you don't use, you don't pay for" [6, p. 121], is near the core of the design of C++. Adherence to this principle is responsible for much of the run-time performance that C++ affords.

Type safety is another core tenet of C++. The benefits of strong type checking have been well-known for several decades. One primary benefit afforded by the C++ type system is early (compile-time) error detection; another is users' ability to devise new types that are then treated on an equal footing with the types pre-defined by the standard.

Both implementation-defined and undefined behaviors make it more difficult to write portable code. It seems clear that minimizing barriers to portability benefits all users.

With this set of guidelines in mind, the Committee is actively evaluating a significant number of proposals for improvements to C++.

A SAMPLING OF PROPOSALS

The standardization process proceeds by considering specific proposals for changes in the standard. Indeed, no

* paterno@fnal.gov
† wb@fnal.gov

[1] As of this writing, there are fifteen national bodies with voting rights in ISO JTC1-SC22/WG21.

progress can be made on an issue unless a specific written proposal is brought forth. In this section, we present a summary of some of the proposals that seem most relevant to the HEP community.

Enhanced Function-Declarations

Run-time performance has been of historic importance to our community. To improve the code they produce, compilers routinely analyze the flow of execution. Current compilers are typically limited to consider only the flow *within* a translation unit.[2] We have brought forth a proposal [7] that would enable compilers to perform comparable analysis between translation units, and to produce improved code.

Consider, in the following code snippet, that the function `f` were implemented in a separate translation unit:

```
void f(double x);

double z(double x, double y) {
  return  f(x) + f(y);
}
```

Unless the compiler is able to perform whole-program analysis (difficult at best and impossible in some cases), it will be unable to perform potentially important call site optimizations.

One example of such an optimization arises in the context of multithreaded environments. In such an environment, it may be possible to schedule simultaneous evaluation of `f(x)` and `f(y)`. When the definition[3] of `f` is not available in `z`'s translation unit, most compilers will necessarily be conservative and, in the interest of safety, forgo this opportunity for improved code generation.

Another example of such an optimization is a form of code factorization often known as *hoisting*. Consider the example:

```
int f( int );

int h( int x ) {
  int i, result = 0;
  while( cin >> i )
    result += i * f(x);
  return result;
}
```

When `f` is expensive to evaluate, it may be reasonable for a compiler to produce code equivalent to:

```
int f( int );

int h( int x ) {
  int i, result = 0;
  bool __first_time = true;
  int __cached_value;
  while( cin >> i ) {
    if( __first_time ) {
      __cached_value = f(x);
      __first_time = false;
    }
    result += i * __cached_value;
  }
  return result;
}
```

Again, however, a compiler must be conservative when it can not see the definition of `f`, because repeated calls to `f(x)` may, in general, yield distinct results.

However, the author of a function like `f` often has the knowledge the compiler lacks—for example, that function `f` is stateless and is free of side effects, and so is safe for concurrent evaluations, for loop hoisting, and/or for similar optimizations. Currently, the language provides no mechanism for the programmer to express this knowledge in a form useful to a compiler.

Our proposal introduces two new qualifiers to address this lack:

1. It allows the programmer to declare a function `pure`, asserting that the function is stateless and commits no side effects.

2. It also allows a programmer to declare, via the qualifier `nothrow`, that a function will not throw any exception, nor otherwise interfere with normal return.

Furthermore, the proposal requires the compiler to verify these declared features when compiling the implementation of any function so declared. Finally, it allows the compiler to make use of these features to produce improved code at each function call site.

Random-Number[4] Toolkit

The generation of high-quality sequences of random numbers is important in many fields, not least in physics. The existing C++ random number generation facility is limited to two functions: `rand` and `srand`. These functions are grossly inadequate for serious use:

- The results are not portable between implementations because the algorithm used for generation is not specified.

- Many implementations exhibit poor "randomness", because conforming implementations can have as short a period as 2^{16}.

- Only one distribution is provided: uniformly-distributed random integers.

[2] A *translation unit* is a source file, augmented by any additional source code that it may directly or indirectly `#include`.

[3] A function's *definition* includes the entire implementation of the function.

[4] Recognizing that what are conventionally called *random* numbers should really be called *pseudo-random* numbers, we shall use the conventional nomenclature.

A proposal [8] before the Committee introduces a far more substantial random number facility.[5]

The proposed random number library provides an extensible toolkit consisting of *engines* and *distributions* for random number generators. An engine may be thought of as a "source of randomness"; each produces a sequence of uniformly-distributed random integers. A distribution creates, from the output of an engine, a stream of random variates with prescribed properties.

The proposal includes a handful of widely-used and high-quality random number engines, as well as a significant number of the most widely-used random number distributions. Further, its modular design makes it easy for users to add their own engines and (perhaps more importantly) their own distributions. Once added, such new engines and distributions will be on an equal footing with those already provided by the library as proposed.

The engines included in the proposal are some of the most widely trusted in the field:

- linear congruential,
- Mersenne twister, and
- subtract-with-carry (also known as *ranlux*).

Engines can also be modified or combined, using either the *discard block* or the *xor combine* algorithms. The output of any engine or engine combination can be used as input to any of the distributions. Furthermore, the output of each engine is guaranteed to be reproducible and portable across implementations.

The proposed distributions are also some of the most important in the field:

- integer uniform, floating-point uniform;
- Bernoulli, binomial, geometric, negative binomial;
- Poisson, exponential, gamma, Weibull, extreme value;
- normal, lognormal, χ^2, Breit-Wigner, Fisher F, Student t; and
- histogram sampling, cumulative distribution function sampling.

These distributions' outputs are guaranteed to be reproducible on a given implementation.

Mathematical Special Functions

C++ support for higher mathematics is largely limited to a small handful of transcendental functions. Recognizing, in part, that this part of the standard library has had almost no attention for quite some time, our proposal [9] to support mathematical special functions (the first significant enhancement to `<math.h>` in circa 30 years) is under active Committee consideration.

As proposed, the special functions library contains many of the most commonly-used functions of mathematical physics:

- Bessel and Neumann functions (cylindrical and spherical, 1^{st} and 2^{nd} kinds),
- Legendre and associated Legendre polynomials,
- Spherical harmonics,
- Hermite polynomials,
- Laguerre and associated Laguerre polynomials,
- Gamma function,
- Complete and incomplete elliptic integrals (1^{st}, 2^{nd} and 3^{rd} kinds),
- Euler beta function,
- Exponential integral,
- Riemann zeta function,
- Error and complementary error function, and
- Hypergeometric and confluent hypergeometric functions.

Standardization brings with it the benefits of quality and reliability, given professional attention to important details sometimes overlooked by application programmers who are focused on their specific problems. Such details include:

- performance (both in *speed* and in *space*),
- correct treatment of corner cases that may need special handling, and
- consistent and appropriate error-reporting and -handling.

Additional benefits include increased portability and reuse. Most importantly, standardization will let our community focus on physics problems rather than on issues related to infrastructure or platform dependency.

The design of the special functions library follows C++ style: special functions are *functions*, as other designs would tend to violate the zero-overhead principle, and/or to be less easily extensible by users who wish to introduce new functions—or new functionality—on the same footing as that provided in the standard.

Slavish adherence to class-based design would suggest an alternative approach, in which each special function would be implemented via a class that contained only `static` member functions: one to evaluate the function,

[5] Fermilab hosted this proposal's author for a week in October 2002. During that time, we were able to communicate our community's needs such that critical elements of the proposal's design were made consistent with our *desiderata*.

one or more to evaluate the function's derivative(s), one or more to evaluate the zeroes of the function, and so on. Such a design uses a class solely as a namespace, but without permitting the extensibility that free functions in a true `namespace` would afford. Since classes are closed to extension, users can not insert new functionality—and clearly no library can anticipate all future needs.

Since C++ is a multi-paradigm language, it does not force us to adhere to any one methodology—it allows us the freedom to choose that methodology which best suits the problem domain. As a bonus, our chosen function-oriented design is compatible with the C programming language[6], and thus is immediately also compatible with most other programming languages.

Shared-ownership Smart Pointers

A *smart pointer*, informally, is an object whose principal responsibility is the management of a memory resource. Today's C++ standard library supplies only one form of smart pointer, `std::auto_ptr`, which has single-owner semantics. However, it has been recognized for some time that a smart pointer with *shared-ownership* semantics is a valuable component.

Unfortunately, no such shared-ownership pointer type is uniformly available today. Since such a capability is often needed, numerous unique variations have been independently invented and re-invented, a situation akin to the days before the standardization of `std::string`. Experts such as Herb Sutter note that implementation of such classes is "exceedingly difficult," especially in the presence of exceptions, so much time is wasted in re-inventing (and debugging) the wheel.

Difficulty of implementation is not the only drawback to the *status quo*: different libraries can not easily communicate when each has its own smart pointer implementation. Since such smart pointers naturally appear in well-designed library interfaces, their omission from the standard has often resulted in contorted designs, in memory leaks, and in other programming artifacts we can describe as "impedance mismatches" between libraries. For all these reasons, and based on years of experience with the very popular `shared_ptr` available from the Boost organization [10]. the Committee is considering a proposal [11] to provide a shared-ownership smart pointer.

Move Semantics

The traditional notion of *copying* a value involves duplicating its bit pattern. Known more precisely as *shallow* copying, this straightforward process is typically applicable to simple data structures only. In contrast, *deep* copying is more often applicable to complex data structures, as it involves recursive copies of contained sub-objects held by pointers, Although deep copying can be an expensive operation, neither deep nor shallow copy operations impact the original value being copied,

In contrast, let us define *moving* as the transfer of the value of an object from a source to a target, with no regard for the value of the source after the move. Moving is typically applicable when the source object:

- will be destroyed shortly after the copy, or
- will get a new value shortly after the copy.

In such cases, moving can be safely used in place of copying, and often at significantly lower cost.

Recognizing that the application of move semantics can lead to considerable performance improvement[7] in common situations, the Committee is considering a proposal [12] to add *move semantics* to the language, thus allowing class authors (and compilers) the ability to choose between moving or copying, depending on the context and its economics.

OTHER DEVELOPMENTS

This paper has concentrated on those proposals to the C++ standards committee that seem of most obvious interest to the physics community. In this section, we briefly mention a few of the other issues under consideration. While some of these items are of less *obvious* importance to the physics community, most are nonetheless important—in many cases because they facilitate implementation of superior libraries.

Decimal Arithmetic

Historically, computer arithmetic has been largely based on binary representations. However, a recently-promulgated draft ISO standard (IEEE 754R) promotes the cause of *decimal arithmetic*. This standard was primarily motivated by financial applications, which often must follow strict legal guidelines for the rounding of monetary values. However, such a development may also be of significant interest to the scientific community.

Vendors have already committed to new hardware in support of decimal arithmetic. There is a long-term view that even suggests that binary arithmetic may ultimately stagnate and fossilize, and that decimal arithmetic may come to dominate numeric types. Accordingly, the Committee is exploring language and library support for decimal arithmetic. This is expected to be a difficult task, as many thorny problems need to be addressed.

Standard Library

The following is a list of selected additional library-oriented proposals under consideration for inclusion in the next C++ standard. Experimental reference implementations for many of these and other proposed library components are freely available from the Boost web site [10].

[6] Indeed, the C standards committees have undertaken incorporation of these same functions.

[7] According to Howard Hinnant, one experimental implementation has seen (in realistic cases) a 10- to 20-fold speed increase.

Unordered associative containers — a proposal to augment the generic containers library with hashed maps and sets.

Regular expressions — a proposal to provide a toolkit for pattern-matching in text strings, incorporating a number of regular expression engines, with special attention to issues of internationalization.

Enhanced function-binders, member-pointer adaptors — two proposals to provide more flexible and uniform mechanisms in support of functional programming[8], to realize more of the benefits of generic programming.

Polymorphic function-object wrappers — a proposal to provide a generalized function pointer (sometimes termed a *callback*) to unify the related but disparate concepts underlying function pointers, member function pointers, and function objects (*functors*).

Tuple types — a proposal to generalize the standard `pair` template, in order to support fixed-size collections (*tuples*) of size greater than two.

Type traits, function-result-type traits — two proposals to facilitate compile-time improvement (via template metaprogramming) of generic algorithms according to the characteristics of the type(s) with which they are instantiated.

Reference wrappers — a proposal to make reference semantics more generally usable; in one important application, standard containers could hold values by reference.

Core Language

The following is a list of selected additional core language-oriented topics and proposals under consideration for inclusion in the next C++ standard.

Dynamic libraries — an effort to standardize the construction and use of dynamic (often known as *shared*) libraries.

Compile-time reflection — an effort to introduce additional compile-time information, so as to make possible more powerful generic code; persistency mechanisms constitute a particularly important application of such a feature.

***Concepts* for generic programming** — a proposal to allow programmers to express the requirements that an algorithm imposes upon its template parameters, and to have the compiler verify that the corresponding template arguments meet those requirements.

[8] *Functional programming* supports the creation and use of functions that operate on, and return, other functions.

Static assertions — a proposal to introduce a compile-time equivalent of the well-known run-time `assert` mechanism.

`decltype` **and** `auto` — a proposal to let the programmer use, in new contexts, a compiler's existing ability to carry out type deduction.

Forwarding constructors — a proposal to reduce code duplication by permitting a programmer to write some of a class's constructors by making use of other constructors of the same class.

Local classes as template parameters — a proposal to extend the use of local (block-scoped) classes to make better use of generic algorithms.

User-defined literals, generalized initializer-lists — two proposals to allow users to express compile-time values for user-defined types in a fashion analogous to that currently available for built-in types.

Null-pointer constant — a proposal to allow programmers to express the distinction between the integer zero and a null pointer.

Template aliases — a proposal to permit a template form of `typedef`.

CONCLUSION

Standard components benefit us all. They require less in-house development and maintenance, they enhance our efforts to share code, and they allow us to focus on physics, rather than on infrastructure.

C++ continues to be of significant interest to the physics community because of its expressiveness, its performance, and our community's significant experience in its use. C++ is now being enhanced in directions of substantive interest to our community, and Fermilab has been actively nudging it in these directions.

We at Fermilab hope to continue supporting the physics community in the international C++ standards effort. We welcome and encourage support from others in the community.

ACKNOWLEDGMENTS

The authors would like to thank the Fermilab Computing Division for supporting their participation in the C++ standardization effort.

REFERENCES

[1] International Standards Organization: ISO/IEC 14882:1998 *Programming Languages – C++*.

[2] International Standards Organization: ISO/IEC 14882:2003 *Programming Languages – C++*.

[3] `http://www.open-std.org/jtc1/sc22/wg21`.

[4] British Standards Institute: *The C++ Standard incorporating Technical Corrigendum 1*. John Wiley and Sons, Ltd., 2003. ISBN 0-470-84674-7.

[5] International Standards Organization: ISO/IEC 9899:1999 *Programming Languages – C*.

[6] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0-201-54330-3.

[7] Walter E. Brown and Marc F. Paterno: *Toward Improved Optimization Opportunities in C++0x* . JTC1-SC22/WG21 paper N1664; same as ANSI NCITS/J16 paper 04-0104. July 16, 2004.

[8] Jens Maurer: *A Proposal to Add an Extensible Random Number Facility to the Standard Library (Revision 2)*. JTC1-SC22/WG21 paper N1452; same as ANSI NCITS/J16 paper 03-0035. April 10, 2003.

[9] Walter E. Brown: *A Proposal to Add Mathematical Special Functions to the C++ Standard Library (version 3)*. JTC1-SC22/WG21 paper N1542; same as ANSI NCITS/J16 paper 03-0125. October 28, 2003.

[10] `http://www.boost.org`

[11] Peter Dimov, Beman Dawes, and Greg Colvin: *A Proposal to Add General Purpose Smart Pointers to the Library Technical Report (Revision 1)*. JTC1-SC22/WG21 paper N1450; same as ANSI NCITS/J16 paper 03-0033. March 27, 2003.

[12] Howard E. Hinnant, Peter Dimov, and Dave Abrahams: *A Proposal to Add Move Semantics Support to the C++ Language*. JTC1-SC22/WG21 paper N1377; same as ANSI NCITS/J16 paper 02-0035. September 10, 2002.