Improving robustness and computational efficiency using modern C++

M Paterno, J Kowalkowski and C Green

Scientific Computing Division, Fermi National Accelerator Laboratory, Batavia, IL, USA

E-mail: paterno@fnal.gov,jbk@fnal.gov,greenc@fnal.gov

Abstract. For nearly two decades, the C++ programming language has been the dominant programming language for experimental HEP. The publication of ISO/IEC 14882:2011, the current version of the international standard for the C++ programming language, makes available a variety of language and library facilities for improving the robustness, expressiveness, and computational efficiency of C++ code. However, much of the C++ written by the experimental HEP community does not take advantage of the features of the language to obtain these benefits, either due to lack of familiarity with these features or concern that these features must somehow be computationally inefficient.

In this paper, we address some of the features of modern C++, and show how they can be used to make programs that are both robust and computationally efficient. We compare and contrast simple yet realistic examples of some common implementation patterns in C, currently-typical C++, and modern C++, and show (when necessary, down to the level of generated assembly language code) the quality of the executable code produced by recent C++ compilers, with the aim of allowing the HEP community to make informed decisions on the costs and benefits of the use of modern C++.

1. Introduction

For nearly two decades, the C++ programming language has been the dominant programming language for experimental HEP. The publication of the current international standard for the C++ programming language [1] has enhanced existing features of the language, and provided several new features, all of which can aid in the writing of code that is more robust, and more easily maintainable, than previously possible. In the field of HEP, however, there is widespread (although not universal) reluctance to make use of some of the features of modern (and even 1998-era) C++. We have found that, in discussions with physicist-programmers, that the most common reason raised against the use of language features that are unfamiliar is the fear of computational inefficiency, when compared to implementing the same or equivalent functionality through use of only the lowest-level features of the language.

In this paper, we address a few of the features for which such fears are groundless, and prove our assertions by comparing either the result of timing alternative implementations, or by comparing the result of disassembling the result of compiling the different implementations. Our goal is to provide the readers with sufficient evidence to overcome their reluctance to use the language and library features we discuss, so that they may benefit from the greater expressiveness of modern C++.

Since our goal is to illustrate how the use of several features of C++ allow the writing of code that is clearer and more maintainable than low-level code, without cost of runtime inefficiency, we have concentrated on very simple uses of these features. This allows us to concentrate on the comparison in question. For example, we compare Fortran-style loop indexing to iterator loops by in the context of a performing a simple sum, so that the comparison of timing results is as sensitive as possible to differences in the loop construction, rather than being dominated by the work done by the loop. This also helps keep the assembly language generated by the compiler as simple as possible, which makes it easier to understand. Sometimes this has the unfortunate side-effect of minimizing the difference in clarity between the low-level coding style and the higher-level facility.

In all the examples, we use the GCC compiler, version 4.8.1, and the compiler options we typically use for production code: `g++ -O3 -std=c++11 -fno-omit-frame-pointer -g`.

2. Loop styles

Modern C++ provides several forms of iteration: Fortran-style index based loops, iterator-based loops, range-based loops, and in some cases generic algorithms that encapsulate the loop. We have found that many people new to C++ avoid the use of all the latter in favor of the first, sometimes for fear of "inefficiency". Listing 1 shows an index-based *for* loop, in a style that is common among beginning C++ programmers.

Listing 1. A non-idiomatic index-based *for* loop.

```
double sum_0(std::vector<double> const& x) {
  double sum = 0.0;
  for (int i = 0; i < x.size(); i++) { sum = sum + x[i]; }
  return sum;
}
```

`sum_0` contains most of the non-idiomatic usages we have seen in loops: repeated calls to `vector::size`, postfix increment of the loop counter `i`, use of separate addition and assignment rather than the `+=` operator, and use of `int` as the type of the loop counter. Of all these, it is the use of `int` as the type for the loop counter that is most often defended as important for efficiency. The idiomatic form for an index-based loop in modern C++ is shown in listing 2:

Listing 2. An idiomatic index-based *for* loop.

```
double sum_1(std::vector<double> const& x) {
  double sum = 0.0;
  for (auto i = 0UL, sz = x.size(); i < sz; ++i) { sum += x[i]; }
  return sum;
}
```

There are several reasons for the idioms in this version of the code. We call `vector::size` once. While this is not critical when the collection we're iterating over is a `std::vector`, because the compiler is able to determine the size of the vector does not change and thus can perform the loop-invariant code motion to remove the repeated calls, for other types of collection the analysis is less simple. We use prefix increment, rather than postfix increment; while this is not critical for a loop counter that is an `int`, for other looping constructs the coast of the postfix increment can be larger than the prefix increment.

The main difference between listing 1 and 2 is the use of `int` rather than `unsigned long` as the type for the loop counter. The C++ standard specifies the argument of the indexing operator for `std::vector` to be of type `std::vector::size_type`, which in the implementation used for compiling these examples, is `unsigned long`. Use of the correct type avoids problems with such things as template type deduction, when use of the wrong type can cause compilation

failures. However, some avoid the use of the correct type because of the belief that use of `int` is more efficient. We timed 1000 repetitions of calls to the `sum` functions, using `vector`s of length 10 million, on an Intel I7 @2.7 GHz processor. The mean and standard deviation for the running time of `sum_0` was 22.98 ± 0.80 million ticks, while that for `sum_1` was 22.80 ± 0.57 million ticks

To see why this is the case, we compare the assembly language generated by the compiler, obtained by dis-assembling the object module produced by the compiler. This allows us to be sure we are observing any effects of optimization steps performed on the assembly language itself. The total size of the code for `sum_0` is 69 bytes; for `sum_1` it is 63 bytes. The sections of code at entry and exit of the functions are the same except for reordering and register renaming. The difference is in the loop constructions themselves, shown in listings 3 and 4.

Listing 3. Assembler for `sum_0`.

```
0030  movq  %rdx, %rcx
0033  movq  %rax, %rdx
0036  leaq  0x1(%rdx), %rax
003a  addsd (%rsi,%rcx,8), %xmm0
003f  cmpq  %rdi, %rax
0042  jne 0x30
```

Listing 4. Assembler for `sum_1`.

```
0030  addsd (%rcx,%rax,8), %xmm0
0035  addq  $0x1, %rax
0039  cmpq  %rdx, %rax
003c  jne 0x30
```

The loop in `sum_0` is slightly larger than that from `sum_1`. For these two loops, the compiler has chosen to use a different instruction to increment the `int` than that chosen to increment the `unsigned long`. As our timing measurements show, there is no significant difference in speed between the two solutions. There is no sign that use of `int` make the code faster, and thus there is neither a speed nor a code size reason to use the non-idiomatic code.

The iterator-based looping construct provides additional advantages: they provide a looping interface that can be used for all types of collections. Combined with the `std::begin` and `std::end` functions, and with appropriate use of `auto` for compile-time type deduction, the resulting code is both clear and flexible, as shown in listing 5.

Listing 5. Iterator-based looping.

```
double sum_2(std::vector<double> const& x) {
  double sum = 0.0;
  for (auto i = begin(x), e = end(x); i!=e; ++i) { sum += *i; }
  return sum;
}
```

Some avoid the use of iterators due to concern about the speed of the resulting code. The time taken by this looping construct is 22.82 ± 0.79 million ticks; this does not differ significantly from the other loops. Listing 6 shows the portion of the assembly language generated for the loop in `sum_2`. It is slightly smaller (because of decreased setup code) than the code for the idiomatic index-based loop, and only 2/3 the size of the code for `sum_0`. The loop itself differs from the index-based loop only in the indexing scheme used in the floating-point addition instruction `addsd`. Concern against the iterator-based loop based on performance is unwarranted.

Listing 6. Assembler for `sum_2`.

```
0020  addsd (%rax), %xmm0
0024  addq  $0x8, %rax
0028  cmpq  %rax, %rdx
002b  jne 0x20
002d  popq  %rbp
002e  ret
```

Listing 7. Range-for looping.

```
double sum_3(std::vector<double>
  const& x) {
  double sum = 0.0;
  for (auto val : x) sum += val;
  return sum;
```

Modern C++ provides an even more compact looping construction, the *range for* loop. This is perhaps the most flexible of the looping constructs in the language, in that it can be used for any sequence that is supported by `std::begin` and `std::end`. Listing 7 shows the implementation of a loop using this construct. The assembly language generated for `sum_3` is identical to that produced for the iterator-based loop. This code is the most clear and flexible of all, and this clarity and flexibility comes with no performance disadvantage at all.

Finally, the C++ Standard Library provides a function template, `std::accumulate` that does the same work as our hand-written loops. The use of this function is shown in listing 8:

Listing 8. Use of the Standard Library algorithm.

```
double sum_4(std::vector<double> const& x)
  { return accumulate(begin(x), end(x), 0.0); }
```

Because it uses an algorithm expressly written for this task, this code is the most compact of all; we would probably call it directly in the place where the sum is wanted, rather than writing `sum_4` at all. The assembly code generated by `sum_4` differs from that generated by iterator-based loop only by the renaming of some registers and the ordering of some of the setup instructions. When a Standard Library algorithm exists that does the required work, it typically produces both the clearest and most maintainable code, and also code as efficient as that which might be hand-written for the same purpose. There is no argument from code size or efficiency to scorn the use of Standard Library algorithms.

3. Lambda expressions

A *lambda expression* is similar to a function without a name, which may declared at "function scope"—within the body of another function. By allowing the placement of the body of a function "in line", rather than requiring the definition of a function outside of a local scope, they make Standard Library-style generic algorithms easy to use. The use of generic algorithms helps make code more uniform and thus easier to read, often makes code briefer and thus more clear. Generic algorithms provide a single point of maintenance for an algorithm, and thus greater chances that improvement in the implementation of an algorithm can improve the performance of many bodies of code. Finally, they ease the introduction of parallel programming libraries, such at Intel Threading Building Blocks[2], which provide parallel algorithms through this style of interface. Many users of C++ are not yet familiar with lambda expressions, and some of those who are familiar with them are hesitant to use them because of concern over the efficiency of the generated code.

Listing 9 shows an example of the sort of loop that can be simplified by the use of one of the Standard Library algorithms.

Listing 9. Hand-written loop to fill a histogram.

```
void fill_hist_1(std::vector<double> const& nums, TH1D& h) {
  for (auto i=begin(nums), e=end(nums); i != e; ++i)
    { h.Fill(*i); }
}
```

Listing 10 shows the use of `std::for_each` to perform the same task, using a lambda expression to produce the function to be called by the `std::for_each` algorithm, and in `fill_hist_3` it shows the use of a named function object created from a lambda expression. This last style is especially useful for two reasons: the same object can be passed to more than one algorithm, and the name of the created object, if well-chosen, documents the behavior of the function.

Listing 10. Using a generic algorithm to fill a histogram.

```
void fill_hist_2(std::vector<double> const& nums, TH1D& h) {
  for_each(begin(nums), end(nums), [&h](double x){ h.Fill(x); });
}
void fill_hist_3(std::vector<double> const& nums, TH1D& h) {
  auto fillhist = [&h](double x){ h.Fill(x); };
  std::for_each(nums.begin(), nums.end(), fillhist);
}
```

Listing 11 shows part of the assembly language produced by the compiler for `fill_hist_2`; to save space, we show only the body of the loop where the `TH1D::Fill` function is called. We note that the line labeled 0x0030 directly calls `Fill`; there is no overhead added by the use of the lambda expression or the generic algorithm. The only difference between the assembly language produced for `fill_hist_1` and `fill_hist_2` is the order of the arguments specified for one of the instructions. `fill_hist_3` generates assembly language identical to that of `fill_hist_2`; this is no sign of the local variable `fillhist`. Thus the enhanced clarity of the code comes at no runtime cost.

Listing 11. Part of the assembly language for `fill_hist_2`.

```
0020  movq  (%r12), %rax
0024  movq  %r12, %rdi
0027  addq  $0x8, %rbx
002b  movsd 0xfffffffffffffff8(%rbx), %xmm0
0030  callq *0x2a8(%rax)
0036  cmpq  %rbx, %r13
0039  jne 0x20
```

4. Bit-fields

One of the lowest-level computing tasks commonly needed in scientific programming is the manipulation of bit-packed data. Such code can be written directly, using bit-wise operators to shift and mask integral data to yield the correct behavior. But such code is less easy to get correct than many expect; it is common to see manipulations that fail to handle all cases correctly, for example by failing to assure that only those bits that should be modified are modified, and no others. Careful use of macros can help, but maintenance of the macros can be needlessly difficult. C++ provides *bit-fields* to aid in such coding. Compare the clarity and maintainability of the code shown in listings 12 and 13, and their uses in listings 14 and 15.

Listing 12. Defining bit-packed data with macros.

```
typedef uint64_t BitWord;
#define MASK01 0x0000000000000001UL
#define MASK10 0x00000000000003ffUL
#define MASK24 0x0000000000ffffffUL
#define BW_APP(p,v,m,s) p = (p & ~(m << s)) | (v & m) << s
#define BW_SET_COUNT(p,v)   BW_APP(p,v,MASK24,00)
#define BW_SET_READING(p,v) BW_APP(p,v,MASK10,49)
#define BW_SET_FLAG(p,v)  BW_APP(p,v,MASK01,63)
#define BW_GET_READING(p) p >> 49 & MASK10
```

Listing 13. Defining bit-packed data with bit-fields.

```
struct BitField { uint64_t count : 24; uint64_t reading : 10;
                  uint64_t flag : 1;};
```

Listing 14. Using the bit-packing macros.

```
BitWord
set_with_macros(uint64_t reading,
                uint64_t count,
                uint64_t flag) {
  BitWord b {0};
  BW_SET_READING(b,reading);
  BW_SET_COUNT(b,count);
  BW_SET_FLAG(b,flag);
  return b;
}
```

Listing 15. Use of bit-fields.

```
BitField
set_with_bits(uint64_t reading,
              uint64_t count,
              uint64_t flag) {
  BitField a;
  a.reading = reading;
  a.count = count;
  a.flag = flag;
  return a;
}
```

The assembly language generated by `set_with_macros` and `set_with_bits` is shown in listings 16 and 17. These listings differ only in the order of the instructions; the size and the speed of the generated code is the same. Again, the improved expressiveness and maintainability of the code come at no cost in size or speed.

Listing 16. Assembler for `set_with_macros`

```
0000  movq  %rsi, %rax
0003  shlq  $0x3f, %rdx
0007  andl  $0x3ff, %edi
000d  andl  $0xffffff, %eax
0012  pushq %rbp
0013  shlq  $0x31, %rdi
0017  orq %rdx, %rax
001a  movq  %rsp, %rbp
001d  popq  %rbp
001e  orq %rdi, %rax
0021  ret
```

Listing 17. Assembler for `set_with_bits`.

```
0000  movq  %rsi, %rax
0003  andl  $0x3ff, %edi
0009  andl  $0x1, %edx
000c  shlq  $0x18, %rdi
0010  andl  $0xffffff, %eax
0015  pushq %rbp
0016  shlq  $0x22, %rdx
001a  orq %rdi, %rax
001d  movq  %rsp, %rbp
0020  orq %rdx, %rax
0023  popq  %rbp
```

5. Variadic templates

The *variadic template*, which allows one to write a single template that works with an arbitrary number of template parameters of arbitrary type, may be the highest-level feature in the language. A single variadic template can be used in place of a whole series of functions, classes, or templates. A good demonstration of the value of variadic templates is in the definition of callback objects. Our goal is to produce a set of callback object to allow code like that in listing 18:

Listing 18. Use of callback objects.

```
int  f1(OneArgSignal s, int a)          { return s.invoke(a); }
long f2(TwoArgSignal s, int a, long b) { return s.invoke(a, b); }
```

To write the `OneArgSignal` and `TwoArgSignal` types lasses directly, we would write two classes, as shown in listing 19. We could instead write a template for each, which would provide the ability to have different *types* as the argument(s) for each `invoke` function, as well as different return types, but we would still need to write two non-variadic templates to handle the different number of arguments. The single variadic template in listing 20 is capable of defining any number of callback types, with arguments of arbitrary type and number. The `using`s introduce aliases which provide convenient names for two of the types produced by this template.

Listing 19. Hand-written callback wrappers.

```
struct OneArgSignal {
  typedef int (*func_t)(int);
  func_t f; // Stored callback.
  int invoke(int a) const
    { return f(a); }
};
struct TwoArgSignal {
  typedef
  int (*func_t)(int, long);
  func_t f; // Stored callback.
  int invoke(int a, long b) const
    { return f(a, b); }
};
```

Listing 20. Template callback wrapper.

```
template <class RT, class... Args>
struct Signal {
  typedef RT (*func_t) (Args ...);
  func_t f; // Stored callback.
  RT invoke(Args... a) const {
    using std::forward;
    return f(forward<Args>(a)...);
  }
};
```

Listings 21 and 22 show the assembly language produced from the use of the callback functions. The code generated from use of the variadic template is exactly the same as that produced for the hand-written types. The variadic template solution is more flexible, more maintainable, and has no size or speed penalty.

Listing 21. Assembler for `f1`.

```
0000  pushq %rbp
0001  movq  %rdi, %rax
0004  movl  %esi, %edi
0006  movq  %rsp, %rbp
0009  popq  %rbp
000a  jmpq  *%rax
000c  nopl  (%rax)
```

Listing 22. Assembler for `f2`.

```
0010  pushq %rbp
0011  movq  %rdi, %rax
0014  movl  %esi, %edi
0016  movq  %rsp, %rbp
0019  movq  %rdx, %rsi
001c  callq *%rax
001e  popq  %rbp
```

6. Conclusion

One of the goals of the design of C++ was to provide a language with "no room below it", that is, to leave no reason to use a lower-level language instead. This goal influenced the design of many of the "higher-level" features of the language, some of which we addressed in this paper. Modern C++ compilers are sufficiently advanced to realize this goal in many cases.

Modern C++ has many features to allow more concise and expressive code that is easier to maintain, when compared with C or with old-style C++. Some of these features take advantage of the type system of C++, and especially the ability to specialize code using templates. Other take advantage of new syntax introduced to the language, often interacting with features of the library. Modern C++ is much more than an object-based language, and provides more abstraction mechanisms than the class and the virtual function. Using language features as they are intended to be used, especially the "high-level" features that are sometimes sources of concern to those unfamiliar with them, can yield code that is both more clear and expressive, and thus easier to maintain, than code written using only the low-level features of the language. As we have shown, in many cases these high-level features introduce no runtime cost, and their is no reason to avoid their use.

References

[1] ISO 2011 Information Technology–Programming Languages–C++ ISO/IEC 14882:2011

[2] Reinders J 2007 *Intel threading building blocks* 1st ed (Sebastopol, CA, USA: O'Reilly & Associates, Inc.) ISBN 9780596514808