Improving robustness and computational efficiency using modern C++

Marc Paterno
Chris Green
Jim Kowalkowski
Fermilab
Scientific Computing Division
CHEP 2013
14 October, 2013
Purpose and Plan

- Identify a few C++ features that some avoid for fear of “inefficiency”, meaning concern about runtime performance.
- Show how use of these features can help improve maintainability and robustness of code, by comparing code that uses low-level constructs to that using the higher-level features.
- Prove there is no reason to fear inefficiency by looking at the output of a commonly-used compiler.

We will do this mostly by looking at the code generated by one modern compiler, GCC 4.8.1, using the compiler options we typically use in development:

```
g++ -O3 -std=c++11 -fno-omit-frame-pointer -g
```
Case 1: looping over the contents of a vector

sum_0, containing the non-idiomatic usages we most often see:

```cpp
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_1, using standard idioms:

- pre-increment counter, += for summing, single call to vector::size
- correct type used for loop index

```cpp
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;
}
```
Case 1: generated code for sum_0 and sum_1

- sum_1, which uses all the standard idioms, is slightly smaller than sum_0.
- Using int is not cheaper than unsigned long.
- Actual iteration done on lines labeled 0030–0042 and 0030–003c.

```assembly
__Z5sum_0RKSt6vectorIdSaIdEE:
0000 pushq %rbp
0001 xorpd %xmm0, %xmm0
0005 movq (%rdi), %rsi
0008 movq %rsp, %rbp
000b movq 0x8(%rdi), %rdi
000f subq %rsi, %rdi
0012 sarq $0x3, %rdi
0016 testq %rdi, %rdi
0019 je 0x44
001b addq $0x1, %rdi
001f movl $0x1, %edx
0024 xorl %ecx, %ecx
0026 xorpd %xmm0, %xmm0
002a jmp 0x36
002c nopl (%rax)
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
0044 popq %rbp
0045 ret

__Z5sum_1RKSt6vectorIdSaIdEE:
0000 movq (%rdi), %rcx
0003 pushq %rbp
0004 xorpd %xmm0, %xmm0
0008 movq 0x8(%rdi), %rdx
000c movq %rsp, %rbp
000f subq %rcx, %rdx
0012 sarq $0x3, %rdx
0016 testq %rdx, %rdx
0019 je 0x3e
001b xorl %eax, %eax
001d xorpd %xmm0, %xmm0
0021 nopl (%rax)
0028 nopl (%rax,%rax)
0030 addsd (%rcx,%rax,8), %xmm0
0035 addq $0x1, %rax
0039 cmpq %rdx, %rax
003c jne 0x30
003e popq %rbp
003f ret
```

4 / 19
Measuring the loop speed

- Measure execution time on AMD Opteron 6136 @2.4 GHz; vector<double> with 10 million values, making 1000 measurements for each algorithm.
- sum_0 is surely no faster than sum_1.
- Using int is not necessarily better than using unsigned long.
Iterators and `std::begin/std::end` provide a looping interface common to all collections (`std::vector`, `std::array`, C-style arrays, ...)

Combined with use of `auto`, code is both clear and flexible

```cpp
double sum_2(std::vector<double> const& x) {
    double sum = 0.0;
    for (auto i = std::begin(x), e = std::end(x); i!=e; ++i)
        { sum += *i; }
    return sum;
}
```
Case 1: generated code for \texttt{sum\_1} and \texttt{sum\_2}

- Actual iteration done on lines 0030–003c and 0020–002b.
- Differences are the indexing mode and the addition reflecting the indexing mode.
- We should not expect to see a difference in execution time.
- The \texttt{vector::iterator} use is efficient because the iterator is implemented as a bare pointer.

\begin{verbatim}
__Z5sum_1RKSt6vectorIdSaIdEE:
0000 movq (%rdi), %rcx
0003 pushq %rbp
0004 xorpd %xmm0, %xmm0
0008 movq 0x8(%rdi), %rdx
000c movq %rsp, %rbp
000f subq %rcx, %rdx
0012 sarq $0x3, %rdx
0016 testq %rdx, %rdx
0019 je 0x3e
001b xorl %eax, %eax
001d xorpd %xmm0, %xmm0
0021 nopl (%rax)
0028 nopl (%rax)
0030 addsd (%rcx,%rax,8), %xmm0
0035 addq $0x1, %rax
0039 cmpq %rdx, %rax
003c jne 0x30
003e popq %rbp
003f ret
\end{verbatim}

\begin{verbatim}
__Z5sum_2RKSt6vectorIdSaIdEE:
0000 movq (%rdi), %rax
0003 pushq %rbp
0004 xorpd %xmm0, %xmm0
0008 movq 0x8(%rdi), %rdx
000c movq %rsp, %rbp
000f cmpq %rdx, %rax
0012 je 0x2d
0014 nopw (%rax)
001a nopw (%rax)
0020 addsd (%rax), %xmm0
0024 addq $0x8, %rax
0028 cmpq %rax, %rdx
002b jne 0x20
002d popq %rbp
002e ret
\end{verbatim}
Comparing three loop constructions

- Measure execution time for same size loops, this time on Intel i7 @2.7 GHz.
- There is no significant performance difference between the iterator-based loop and the indexing-based loop.

![Graph showing time per loop for sum_0, sum_1, and sum_2](image-url)
Case 1 continued: even better looping constructs

- C++11 introduces *range-for* loops, and
- There is a standard library algorithm that does this work.

```c++
1 double sum_3(std::vector<double> const& x) {
2   double sum = 0.0;
3   for (auto val : x) { sum += val; }
4   return sum;
5 }
```

```c++
1 double sum_4(std::vector<double> const& x) {
2   return std::accumulate(x.begin(), x.end(), 0.0);
3 }
```

- Each of these produces assembly code equivalent to that of `sum_2`, the iterator-based loop.
- They differ only in assignment of registers or ordering of instructions.
Case 2: Lambda expressions

A lambda expression is like a function without a name. They make the Standard Library algorithms easy to use:

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

void fill_hist_2 (std::vector<double> const& nums, TH1D& h) {
    std::for_each (std::begin(nums), std::end(nums),
                   [&h](double x){ h.Fill(x); });
}
```
Case 2: generated code for fill_hist_2

The call to TH1D::Fill is on line 0030.

The standard library algorithm pattern also lends itself to various parallelism libraries, e.g. Intel TBB.

- There is no runtime artifact from the lambda-expression.
- The only differences between the assembly code for fill_hist_1 and fill_hist_2 is the order of the arguments of cmpq on line 0017, and the names of the functions.
- Using a standard library function with a lambda is as efficient as the best hand-written loop, and provides no chance to for a mistake is the loop construction.
Case 3: Manipulation of bits, the lowest-level task

C++ provides **bit-fields** for cleaner code than the more error-prone shifts and bitwise ands and ors.

- Careful writing yields readable use of macros.
- But the macros themselves are hard to maintain.
- Compare the ease of identifying how many bits are used by each element, and the verbosity of the code.

```cpp
typedef unsigned long BitWord;
#define MASK01 0x0000000000000001UL
#define MASK08 0x00000000000000ffUL
#define MASK09 0x00000000000001ffUL
#define MASK10 0x00000000000003ffUL
#define MASK24 0x00000000000000000000000fffUL
#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_VERSION(p,v) BW_APP(p,v,MASK08,24)
#define BW_SET_TYPE(p,v) BW_APP(p,v,MASK08,32)
#define BW_SET_SEQ(p,v) BW_APP(p,v,MASK09,40)
#define BW_SET_READING(p,v) BW_APP(p,v,MASK10,49)
#define BW_SET_FLAG_1(p,v) BW_APP(p,v,MASK01,59)
#define BW_SET_FLAG_2(p,v) BW_APP(p,v,MASK01,60)
#define BW_SET_FLAG_3(p,v) BW_APP(p,v,MASK01,61)
#define BW_SET_FLAG_4(p,v) BW_APP(p,v,MASK01,62)
#define BW_SET_FLAG_5(p,v) BW_APP(p,v,MASK01,63)
#define BW_GET_READING(p) p >> 49 & MASK10

struct BitHead {
    unsigned long count : 24;
    unsigned long version : 8;
    unsigned long type : 8;
    unsigned long seq : 9;
    unsigned long reading : 10;
    unsigned long flag_1 : 1;
    unsigned long flag_2 : 1;
    unsigned long flag_3 : 1;
    unsigned long flag_4 : 1;
    unsigned long flag_5 : 1;
};
```
Case 3: using the macros and bit-field

```c
BitWord set_with_macros(unsigned long reading,
                        unsigned long count,
                        unsigned long flag_5) {

    BitWord b {0};
    BW_SET_READING(b, reading);
    BW_SET_COUNT(b, count);
    BW_SET_FLAG_5(b, flag_5);
    return b;
}

BitHead set_with_bits(unsigned long reading,
                       unsigned long count,
                       unsigned long flag_5) {

    BitHead a {0};
    a.reading = reading;
    a.count = count;
    a.flag_5 = flag_5;
    return a;
}
```
Case 3: comparison of generated assembly language

- The only difference in the generated code is the order of instructions.

<table>
<thead>
<tr>
<th>__Z15set_with_macros:mmm:</th>
<th>__Z13set_with_bits:mmmm:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 movq %rsi, %rax</td>
<td>0000 movq %rsi, %rax</td>
</tr>
<tr>
<td>0003 shlq $0x3f, %rdx</td>
<td>0003 andl $0x3ff, %edi</td>
</tr>
<tr>
<td>0007 andl $0x3ff, %edi</td>
<td>0009 pushq %rbp</td>
</tr>
<tr>
<td>000d andl $0xffffffff, %eax</td>
<td>000a andl $0xffffffff, %eax</td>
</tr>
<tr>
<td>0012 pushq %rbp</td>
<td>000f shlq $0x31, %rdi</td>
</tr>
<tr>
<td>0013 shlq $0x31, %rdi</td>
<td>0013 movq %rsp, %rbp</td>
</tr>
<tr>
<td>0017 orq %rdx, %rax</td>
<td>0016 shlq $0x3f, %rdx</td>
</tr>
<tr>
<td>001a movq %rsp, %rbp</td>
<td>001a orq %rdi, %rax</td>
</tr>
<tr>
<td>001d popq %rbp</td>
<td>001d popq %rbp</td>
</tr>
<tr>
<td>001e orq %rdi, %rax</td>
<td>001e orq %rdx, %rax</td>
</tr>
<tr>
<td>0021 ret</td>
<td>0021 ret</td>
</tr>
</tbody>
</table>

- The superior maintainability has no runtime cost.
- The low-level language feature is not more efficient than the high.
Case 4: Encapsulating callbacks using variadic templates

- The variadic template may be the highest-level abstraction facility in C++; it allows templates with arbitrary numbers of arguments.
- These can replace whole families of hand-written functions.

We want the use of our callbacks to look like:

```c++
int f1(OneArgSignal s, int a) {
    return s.invoke(a);
}

int f2(TwoArgSignal s, int a, double b) {
    return s.invoke(a, b);
}
```
Case 4: Hand-written callback types

```c
// One type per function signature is required.
struct OneArgSignal {
    typedef int (*func_t)(int);
    func_t func; // Stored callback function.
    int invoke(int a) const { return func(a); }
};

struct TwoArgSignal {
    typedef int (*func_t)(int, double);
    func_t func; // Stored callback function.
    int invoke(int a, double b) const
        { return func(a, b); }
};
```

We could also do this with templates, but we need a different template for each number of arguments.
Case 4: Callback types from a variadic template

```cpp
// One template handles all function signatures.
template <typename ResultType , typename ... Args>
struct Signal {
    typedef ResultType (∗func_t) (Args ...);
    func_t func; // Stored callback function.
    ResultType invoke(Args... args) const {
        return func(std::forward<Args>(args) ...);
    }
};

// The alias declaration introduces a typedef name
using OneArgSignal = Signal<int , int >;
using TwoArgSignal = Signal<int , int , double >;
```

The single template can be used for any number of arguments, of any types.
Case 4: Generated code for f1 and f2

The code generated for the variadic template is exactly the same as for the hand-written types.

Note that with this technique we don’t even get a call to the function: we jump directly to it.

The variadic template allows much more concise code, that is more powerful, and generates assembly code as efficient as the hand-written types.
Conclusion

One goal 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 talk.

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.

So: Write code for clarity and maintainability, using “high-level” features as they are intended, without worry about runtime efficiency.