presented by Krzysztof Genser
for Fermilab Geant4 performance group
14th Geant4 Workshop, October 2009

On improving Geant4 performance, robustness and easing code maintenance

Members of Fermilab Geant4 (G4) Performance Group

- Walter Brown*
- Mark Fischler
- Krzysztof Genser*
- Jim Kowalkowski
- Marc Paterno
- Ron Rechenmacher
  - For a total of about 0.6 FTE
  - reviews this year done by people denoted with *

Talk Outline

- Fermilab Geant4 (G4) Performance Group Activities and related matters
- Suggestions/Reminders regarding C++ coding techniques/style inspired by recent reviews

Past year's activities

- Completed Reviews of CHIPS and Field Propagation Modules
  - CHISP g4 version 9.1.p03 FPM: g4 version 9.2.p01
  - Delivered the results to the G4 Management in the form of written reports
- Nature of the findings:
  - Most of the comments were related to the C++ coding techniques having impact on code robustness and maintenance
  - Among other findings: a potential ~0.5% level timing improvement in CHIPS in a stand alone test resulting from replacing a collection of pointers to objects with collections of objects (potentially ~0.5% per collection?)
    - This was in addition to the previous finding where
      std::vector<G4Double>* T was replaced with
      std::vector<G4Double> T which resulted in ~1.5% timing improvement in the CMS offline environment
  - (No significant opportunities for timing improvement noted in Field Propagation Module)

Profiling/Tools Used

- Reviews were guided by profiling done both using standalone Geant4 tests/examples and in the context of CMS environment CMSSW (cmsRun)
- Tools used to guide the reviews:
  - "SimpleProfiler" (C++ dynamic library collecting detailed call stack samples, uses “libunwid”; about 1% overhead)
  - "Performance Data Base" (see next slide)
    - esp. helpful in simplifying the assessment of the statistical significance of the timing differences
  - Valgrind's Tool Suite

More on past year's activities

- Continued to improve "Performance Data Base"
  - our system (written using Ruby on Rails and mySQL) for recording performance run data
  - work in progress continued to be done with the help of student interns from Northern Illinois University
- Explained a previously found event irreproducibility
  - correlated with bifurcation of event processing times across many of the same jobs and
  - different amount of random numbers drawn from generator depending on computer architecture
  - traced to different "firmware" implementations of sin (or, in general, transcendental) functions for different CPU brands

Suggestions/Reminders regarding C++ Coding Techniques/Style

Suggestions/Reminders regarding C++ coding Techniques/Style

- Remarks inspired by and generalized from the reviewed G4 code
  - many of us came to C++ from other languages or learned C++ when it was in its early deployment years
  - the remarks below are partially a result of the evolution of C++ and coding techniques
- Coding techniques can impact not only performance but also the effort needed to read the code and therefore the cost required to maintain it

Main Suggestions

- Make sure to completely construct/initialize and copy objects
- Move from native pointers and arrays to higher level objects ("domain-specific abstractions")
- Utilize Standard Library (C++StdLib) more: use, as often as feasible, not only numerical functions but also containers and algorithms

Objects' State and basic behavior

- Make sure to completely construct/initialize and copy objects
  - one should be able to "reason about the code", i.e. constructors and assignment operators should behave in the way most programmers would expect
  - one should be able to use the objects with C++StdLib containers and algorithms which do have certain requirements (see e.g., N.M.Josuttis, "The C++ Standard Library"):
    - public, "faithful" copy constructor & assignment operators and public, nonthrowing destructor;
    - some containers and algorithms may also require default (no argument) constructor, equality ("==") operator, "<" operator

More on construction and copying

- Use initializer lists in the constructors (for performance)
  - list data members in the order they are defined (for consistency)
- If there are no reasons to copy instances of a class the class should be non-copyable
  - declare its copy functions private and leave them unimplemented

Copy constructors and assignment operators

- Do not define/declare copy constructor/assignment operators when not needed (e.g., for objects with no pointer data members or no data members at all) letting compiler generate them
- In cases where the compiler-generated constructors/assignment operators would not be adequate, follow the prescription shown in e.g., H. Sutter "Exceptional C++" using copy-and-swap as the implementation technique of the assignment operator

Pointers == Implementation Details

- Avoid exposing pointers (of any kind) in interfaces whenever possible
  - especially avoid exposing native pointers to clients as the pointee ownership becomes unclear
  - it is easy to break code which uses pointers
  - code with pointers is more difficult for compilers to optimize

Containers and Pointer Use

- Prefer containers of objects over containers of pointers; limit use of containers of pointers to the following cases:
  - when the exact type of the pointees is unknown (when relying on polymorphic behavior)
  - when one needs to perform operations which would involve expensive data movement (e.g., swapping data when performing sorting)
    - btw, swapping std::vectors (of anything) is not expensive [O(1) (constant time)], no matter their size
  - data replication would occur (e.g. creation of multiple lists of the same objects: shared pointees)
- Operations on containers of pointers are more involved:
  - each traversal introduces an extra dereferencing operation
  - each time such a container were to be created, copied or destroyed, it incurs the overhead of more expensive dynamic memory management

Smart vs. Native Pointers

- Smart pointers automate the management of resources
  - they relieve programmers from the burden of doing so and ensuring that the management is not accidentally overlooked or mismanaged
- Prefer smart over bare/native pointers, e.g., std::auto_ptr and, to be included in the new C++ standard, unique_ptr and shared_ptr
  - unique_ptr and shared_ptr are available already in, e.g., gcc by enabling the correct compiler flags; (there are other sources e.g. boost library)
  - remember though that std::auto_ptrs can not be used as container elements (they do not have copy semantics which C++StdLib containers expect)

More on Standard Library Containers/Algorithms

- Prefer to use appropriate containers (usually std::vector) from the C++StdLib instead of arrays (together with C++StdLib algorithms):
  - one can replace explicit (hand-coded) loops to initialize or to copy arrays with std::copy, std:fill, etc…, e.g.,

    ```
    std::fill_n(Array1+0,mySize,myValue);
    std::copy(Array1+0,Array1+mySize,Array2+0);
    ```
  - when using std::vector, its copy and assignment operators replace the explicit operations:

    ```
    v2=v1
    ```
- Using even such simple algorithms as std::min and std::max improves the clarity of the program text, and often provides performance benefits as well (see an example later)

Functors (function objects) and Standard Library

- Using Functors (function objects) with C++StdLib algorithms opens more possibilities and allows for more compact and efficient code
  - Functor is an object behaving like a function or "object which can be invoked with ( )" (function call operator)
- Simple functor example:

```
class multiplyBy {
private:
    double multiplier;
public:
    explicit multiplyBy(double m) : multiplier(m) { };
    void operator( ) (double& a) {                    // note the operator( )
       a *= multiplier;
    };
};
```

Functors (function objects) and Standard Library cont'd

- Using the functor from the previous page:

```
…
std::vector<double> v;          //container of  doubles
…
std::for_each(v.begin(), v.end(), multiplyBy(factor));
…
```

  - for_each takes an instantiated multiplyBy object and calls its operator() for each element of v
    - factor value can be determined at the run time

Functors (function objects) and Standard Library cont'd

- Templatized functor example:

```
template <typename T1, typename T2>
class multiplyBy {
private:
    T2  multiplier;
public:
    explicit multiplyBy(T2 m) : multiplier(m) { };
     void operator( ) (T1& a) {          // note the operator( )
       a *= multiplier;
     };
};
```

Functors (function objects) and Standard Library cont'd

- Using the functor from the previous page:

```
…
std::vector<MomentumVector> VMV; //container of momentum
   vectors
…
std::for_each(VMV.begin(), VMV.end(),
   multiplyBy<MomentumVector,double>(factor));
```

- Functors have advantages over functions
  - functors can be initialized/modified during run time
  - multiple operations are usually faster compared to function operations
    - functors are usually inlined as the compiler typically has more information compared to the case when using pointers to functions

Class Invariants and Debugging/Testing

- Consider to explicitly spell out class invariants (state of the object) and encode them as functions which could be used in debugging and unit testing (to check if the object is in a consistent state)
  - the functions could be removed with #ifndef/#endif blocks in the production code
    - (see e.g., B.Stroustrup The C++ Programming Language 24.3.7.1)
  - Some classes have very "natural" invariants, e.g., an invariant mass for a particle four momentum

Loops

- prefer pre-test loops: while…, for… over post-test loops: do…while
  - do…while does not permit zero number of executions of the loop body
- use proper type for the loop controlling variables
  - to minimize conversions
- prefer "!=" operator in loop predicates
  - not all iterators support operator "<"
- (prefer pre-increment/decrement operators when the returned result of the post-increment/ decrement is unused: ++a vs. a++)

Keyword explicit

- Constructors callable with a single argument are the so called *conversion constructors*
  - They can be used implicitly, e.g., in assignments:

```
B b = aConstantOfTypeB;
A a = b; //implicit automatic type conversion
```

- They should be declared "explicit" when appropriate, to avoid sometimes unexpected or unintended conversions:

```
explicit MomentumVector(double px=0.0,double py=0.0, double pz=0.0)
    : mvpx(px), mvpy(py), mvpz(pz) { };
ParticleMV = 1;
//without the keyword explicit it means:
MomentumVector ParticleMV = MomentumVector(1.0,0.0,0.0);
```

and not e.g., a unit vector with equal components etc…

Automatic vs. dynamic allocation

- For local variables prefer automatic (stack) over dynamic (heap) allocations
  - remember the earlier-mentioned result of the replacement of

    std::vector<G4Double>* T

    with

    std::vector<G4Double> T

Style

- Provide the inline documentation (== comments); at the minimum clearly define the mission of the classes/functions and state each algorithm's name or main idea
- Prefer short over long functions with many lines of code, preferably with tens not (many) hundreds of lines.
- Consistently and consequently name literals (i.e., provide meaningful names for constants used within the code) to enable readers to understand the purpose of a constant and to ease future code maintenance

Declaring/initializing Objects

- Declare objects when they are ready to be initialized (or as late as possible)
- Initialize them with their correct values (rather than providing a value that may have to be changed almost at once)
- Use the ternary ?: operator when appropriate and C++StdLib functions when available, see the examples below

```
//example1
T g = z;
if (y>z)
   g = y;
```

compared to:

```
T g = std::max(z,y);
```

```
//example2
T g;
g = z;
if (x>0)
   g = y;
```

compared to:

```
T g = x>0 ? y : z;
```

Policy regarding .hh & .cc files and inlining

- Consider eliminating .icc files (leaving just the .hh & .cc files) and adopting a more coherent policy regarding inline declarations, preferably in accordance with the DRY (Don't Repeat Yourself) principle, e.g., by defining inline functions where they are declared in the *.hh file (particularly as such functions tend to have very short definitions)
- Having two files to look at instead of three would help in locating comments and require that only .hh file is to be looked at to find out which function is declared inline

Summary/Main Suggestions

- Fermilab Geant4 performance group have continued to profile and review the Geant4 code
- Main suggestions inspired by the reviews/profiling:
  - Make sure that objects (all data members) are completely initialized and that the copy constructors and assignment operators do complete copies as expected by the Standard Library Standard Library Containers and Algorithms
  - Move from native pointers and arrays to higher level objects including Standard Library Containers (also to enable wider use of related Algorithms)
  - Provide more inline documentation (== comments); at the minimum always clearly define the mission of the classes/functions
- Plans:
  - Considering to continue reviews and profiling (concentrate not only on C++ itself but also more on the algorithmic level)
  - Continue to develop and improve our "Simple Profiler" and "Performance Data Base" tools

Suggested implementation of assignment operator – backup slide

- Ensure that T has a correct and faithful copy constructor, and a non-throwing destructor that correctly disposes of any resources held by the class
- Provide T with a non-throwing *swap* function to exchange the values of two variables of type T.
  - It is always possible to implement such a swap function for any class T by invoking an appropriate swap function for each of T's data members: For each data member of T whose type is a native (built-in) type, invoke std::swap, and for each data member of T whose type is either a library type (e.g., std::vector) or a user-defined type, invoke its own swap member.
- Write T's copy assignment operator according to the following model which, by construction, is correct as well as exception-safe in all cases, including the rarely-occurring self-assignment (see e.g., H.Sutter "Exceptional C++")

```
T & operator = ( T const & other ) {
   T tmp( other );    // ← if this throws it is before the left hand side is affected!
   tmp.swap(*this);
   return *this;
}
```

- (Ensure operator "==" ,if provided, holds true after copy)