# Findings and Recommendations Regarding the GEANT4 CHIPS Model

W. E. Brown and K. Genser
*Fermi National Accelerator Laboratory*
2009-01-28

## Contents

## 1 Introduction

### 1.1 Scope of assessment

Our charge in reviewing GEANT4 CHIPS module was to "Assess the CHIPS module in Geant4 in order to

- identify opportunities for savings of CPU time and memory footprint
- note aspects and classes whose purpose is not clear or design/implementation is confusing
- provide suggestions for key opportunities for simplification of its structure, and decoupling of capabilities."

We observed that this module (as distributed in G4 version 4.9.1.p03) consisted of more than 76,000 lines of code, distributed over 140 source files (headers + implementation files). Since we could not possibly look at all of this code in the time allotted for the review, we performed some initial measurements[1] on examples[2] provided with the code in order to determine which parts of the module are in heaviest use. We then concentrated on the heavily-used classes and on the classes providing the interface to the CHIPS module, with occasional (brief) excursions into other classes.

This review focuses on technical issues regarding the module's implementation in C++ code. It may be appropriate, once the issues raised below have been addressed, to task another review team with the charge of assessing the module's class design and any other high-level areas of concern. For example, we noted in the commentary and in the code numerous signs of the module's historical evolution. A reconsideration of the current design with the benefit of hindsight may yield further improvements.

### 1.2 Outline

We present, first, some general observations. Along with each item, we indicate our assessment (1) of the item's relative importance and (2) of the relative difficulty of remedying the issue. In each case, we use codes H = high, M = medium, or L = low at the end of each item. The order of the observations provides a first approximation to the order in which we recommend that they be considered for implementation: H/L, M/L, H/M, L/L, M/M, H/H, M/H, L/M, L/H.

In the subsequent section, we present, organized by class in alphabetical order, remarks specific to individual classes and their members. The order of these observations and remarks does not necessarily correspond to their order of importance.

We then describe an experiment we undertook to gain insight into the feasibility of modifying the CHIPS module code, followed by one possible sequence for carrying out the modifications recommended in this review. We conclude the paper with a few acknowledgments.

## 2 General observations

### 2.1 Code documentation

**Version labeling.** We note that only some of the source files include an internal cvs tag. [M/L]

[1]Via `valgrind`/`callgrind`.

[2]First `test35` and then novice example `N04` (modified to use CHIPS via the QGSC model instead of QGSP).

**Naming similarities.** We observed that names of distinct classes are often very similar, and it is unclear how or even whether the classes are related, nor which ought be used under which circumstances. As one case in point, we note the existence of classes: `G4StringChipsInterface`, `G4StringChipsParticleLevelInterface`, and `G4QStringChipsParticleLevelInterface`. Given such similar names, it is unclear that all are needed, nor how they are distinguished in purpose and thus in practice. (Perhaps there is external documentation that already addresses this need, but we have not yet found it.) [H/H]

**Inline documentation.** In browsing the code, we noted the general absence of internal documentation for maintainers. For example, it is a recognized best programming practice to provide certain introductory information for each class and for each non-trivial function. We would expect to find at least a statement of design intent, *i.e.*, what the class or function "does for a living" (its principal purpose, often known as a *mission statement*). For the most part, the CHIPS module seems to provide such narrative only occasionally. [M/H]

**Invariants.** It is another recognized best practice to provide each class with an invariant (known in this context as a *class invariant*). The purpose of invariants in general is to assist in reasoning about the code, especially in assuring the code's correctness. Unfortunately, we found no such invariants, neither at the class level nor at any lower level. [M/H]

## 2.2 Provisions for testing

We were unable to locate a framework for comprehensively testing the types and functions comprising the CHIPS module. In the absence of such a test suite, it will be difficult to verify the correctness of any non-trivial modifications that may be made to the module's code.

## 2.3 Coding practices

**Incomplete initializations.** We discovered a common pattern in a large number of the constructors and assignment operators that we examined: some fraction of the data members are left uninitialized or uncopied. This is contrary to recommended practice, as it makes it unnecessarily difficult to reason about code that uses such constructors. It also introduces unnecessary fragility and coupling in the code, as one must be careful to avoid calling a function that uses an uninitialized data member before calling a function that initializes them. Further, in the case of copy constructors, it is a violation of the usual copy semantics that the result of a copy operation be in fact a faithful reproduction of the original. Such violation is both risky and error-prone, and certainly complicates program maintenance. If the functionality currently provided by these incomplete copy functions is strictly needed by clients of the existing code, then this functionality will need to be provided in a new function by a different name and their clients updated to call these new functions. [H/L]

**Dynamic instead of automatic memory allocation.** We found several occurences in which a local variable is needed, but instead of simply creating the variable automatically (*i.e.*, on the runtime stack), the CHIPS module creates the variable on the heap and creates on the stack a pointer to that heap object. This practice also adds unnecessary cost and complexity. [H/L]

**Aliasing.** In some places, pointers are used as simple handles to existing data, an implementation of a technique often termed *aliasing*. Aliases typically do not share in the ownership of the underlying object, and thus have no responsibility with respect to that object's lifetime. In modern C++, such aliasing is typically expressed via the use of reference types rather than pointer types. The advantage of using references in this way is that a reader can quickly identify the purpose of the code, and not need to puzzle out what role is served by the pointer. [H/L]

**Implementation of copy assignment.** We noted in many classes the complexity of their copy assignment operator implementations. These functions largely parallel their respective classes' copy constructors, generally augmented by a special test to avoid self-assignment (which rarely, if ever, is used in practice). Modern C++ recommends, for any class `T`, the following approach instead: First, ensure that `T` has (1) a correct and faithful copy constructor, and (2) a non-throwing destructor that correctly disposes of any resources held by the class. Second, provide `T` with a non-throwing `swap` function to exchange the values of two variables of type `T`.[3] Finally, 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). [H/L]

```
T &  operator = ( T const & other )  {
  T  tmp( other );
  swap( tmp );  // assumes swap is a member function
  return *this;
}
```

**Increment operator selection.** Best practices regard the unnecessary use of postincrement in place of preincrement operations as a code "pessimization." In loop control and also occasionally elsewhere in the code we saw consistent use of the postincrement operator, and saw no uses at all of the preincrement operator. [M/L]

**Initializer lists.** Many constructors found in the CHIPS module follow another common pattern: their initializer list is incomplete (or occasionally even nonexistent). Such omission can lead to uninitialized data members as described above, or to reduced performance due to doubly-initialized data members. [M/L]

**Const-correctness.** While `const`-qualification is used in the code (especially to qualify accessor member functions), we noted additional opportunities for its use. For example, an alias used only to access (and not modify) an object should typically be `const`-qualified so that a reader can quickly understand the role of the aliasing variable. Additionally, it was confusing to see non-`const` contents in a container that was declared `const`. [M/L]

**Unnecessary constructors.** We found several instances of a constructor labeled as a "copy constructor by pointer." C++ does not recognize such a function as a copy constructor; instead, the function is considered a "conversion constructor." Not only does the existence of such functions introduce unnecessary duplication in the program text, they are potentially harmful when used. The following hypothetical declaration illustrates one of the risks:

```
T t( new T );  // implicit T* => T conversion leads to a memory leak!
```

The only real benefit of such functions is to avoid typing an extra asterisk (`*`) in each of the few places where such a function is called. [H/M]

**Non-`explicit` conversion constructors.** We observed no use of the `explicit` keyword. It is generally recommended that this keyword be applied to all conversion constructors to prevent the compiler from performing the conversion unless explicitly requested to do so. The absence of this keyword can lead to unexpected and unintended conversions, making it more difficult to understand and reason about code. [H/M]

[3]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.

**Pointers' frequency.** The CHIPS module was observed to traffic extensively in native C++ pointers. Such pointers are widely considered to be a very low-level data structure that, while effective, are sources of significant complexity in design and of bugs in implementations. Modern C++ recommends instead the use of smart pointers and similar handle types. The Standard Library, for example, provides `std::auto_ptr` and will in the near future provide additional smart pointers (*e.g.*, `shared-ptr` and `unique_ptr`) that have for almost ten years been available from other sources. [H/M]

**Pointers in containers.** We found that the CHIPS module extensively uses containers of pointers to objects, rather than containers of the objects themselves. There are a few good reasons to use containers of pointers; for example (1) when the pointees are to be shared and thus avoid the overhead of multiple copies of them, or (2) when the exact type of the pointees is of an unknown derived type and the `virtual` mechanism is to be used by pointing to instances of the base type. However, neither of these reasons seemed to apply here, and in their absence the chosen approach is a source of potentially significant performance degradation, as each traversal of the container introduces an extra dereferencing operation. Further, each time such a container is created, copied, or destroyed, it incurs the overhead of expensive dynamic memory management, together with its attendant added complexity. [H/M]

**Unnecessary recalculation.** In certain loops traversing but not changing a container such as an `std::vector<>`, we see that the `size()` of the container is recalculated on each iteration despite the fact that the size does not change from one iteration to the next. Such unnecessary re-evaluation per iteration seems therefore unnecessary and expensive. [L/L]

**Guarded loops.** We observed that a very large fraction of all `for`-loops in the CHIPS module are preceded by an `if` such that the loop is initiated only when the `if`'s predicate is `true`. Much of the time, we found this predicate to be isomorphic (or even identical) to the predicate provided with the loop itself. Since the number of iterations of a C++ `for`-loop is permitted to be zero, the presence of such an `if` is superflous, decreasing the code's readability, increasing slightly its compilation costs, and (depending on compilation details) negatively impacting the code's performance. [L/L]

**Loop predicates.** Modern C++ programming techniques recommend the use of loop predicates involving the `!=` operator rather than, say, the `<` operator. There are several reasons for this preference: For one, such use permits a stronger post-loop assertion; for another, loops controlled by iterators must already use `!=` in their predicates since iterators are generally not required to support `<`. The bulk of the predicates we encountered in the CHIPS module use the `<` operator. [L/L]

**Code appearance.** The indentation used by the text of the CHIPS module is inconsistent at times. Much of the code is simply difficult to read, devoid of conventional spacing between tokens and similar linguistic niceties. Comments frequently contain misspellings, leading to occasional doubt as to the underlying intent. [L/L]

**Pointers to implement optional arguments.** Another use of pointers in the CHIPS module is to implement a function parameter that is optional. When the corresponding argument is omitted, the pointer value defaults to the null pointer values; with the argument is to be provided, it is passed via a pointer. Such an idiom is far from transparent, and can be better implemented via a class designed for such a purpose. Such classes have been publicly available for a number of years, and do not use pointers in their implementation. [L/L]

**Provisions for debugging.** The presence of copious intrusive debugging statements (some of which seem to introduce side effects!) frequently makes the logic of the underlying code significantly more difficult to read due to the extra bulk. Further, since the debugging statements are guarded by an assortment of symbol definitions, it would be useful to have documentation as to these symbols' meanings. [M/M]

**Inadequate use of the Standard Library.** We observed only minimal use of the general facilities of the C++ Standard Library. For example, we saw explicit (hand-coded) loops to initialize or to copy arrays that could more simply have been coded as simple calls to such Standard Library functions as `std::copy`, `std::uninitialized_fill`, *etc.* Using the Standard Library improves the clarity of the program text, and often provides performance benefits as well. Additionally, we observed explicit looping over arrays where the entire loop could be avoided via the use of `std::vector` operators such as copy assignment. [H/H]

**Unusual copy semantics.** In addition to the above observations regarding copy operations that copy only selected data members, we observed certain copy operations that deliberately inject known values (constants) into the putative copy. This is another source of potential complication. [H/H]

**Unusual equality semantics.** Many of the classes we inspected provide operators to determine equality or inequality of objects of their type. However, the CHIPS module consistently uses a non-standard definition of equality: two objects are considered equal if and only if they are identical. Such a definition makes it impossible to compare arbitrary objects of the type, as even the result of a standard copy can't compare equal to its original. [H/H]

**Function bulk.** The CHIPS modules exhibits several examples of functions of extreme size. Several such functions are of order 3000 lines long, leading to classes whose implementation files are more than twice as large. Other functions are of size measuring in the hundred of lines. Good code hygiene considers such function sizes as excessive. It is essentially impossible to reason effectively in the presence of such bulk, and hence making any updates becomes fraught with danger. [H/H]

**Code structure.** We noted code that seems awkwardly structured, leading to unnecessary compilation expenses as well as to impediments to understanding. [H/H]

**Literals.** It has long been recommended practice, in C++ as well as in other programming languages, to provide meaningful names for constants used within the code. This enables readers to understand the purpose of a constant, with no need to infer its intent from its context. The code we inspected only occasionally followed this practice. In particular, we note that literals are often used instead of Geant4-defined standard particle names. Moreover, when Geant4 does not provide suitable codes for particles specific to the CHIPS module, we observed further use of literals rather than meaningful names. Finally, we observed some (occasional) use of named constants, followed closely by literals for the same values. [M/H]

**Inheritance.** We found in the CHIPS module certain inheritance relationships that conform poorly to current best practices. There is one instance of inheritance from `std::vector<>`, but that template is not designed to serve as a base class. There are also public inheritance relationships that seem to be for the purpose of inheriting implementation (behavior only) rather than to obtain the *is-a* relationship that is the usual purpose of public inheritance. [M/H]

**Loop control variables' types.** The majority of the loops in the CHIPS module appear to be loops controlled by a counting variable. We observed no consistent policy regarding the choice of type for such variables, and often saw inconsistencies between the type declared for such a variable and the type of the value with which it is initialized or against which it is compared. Each such inconsistency requires a conversion to resolve, and so negatively affects performance. [L/M]

**Early declarations.** In some programming languages, variables must be defined at or near the beginning of a function. In C++, it is possible to define variables at any point within a function, and it is standard practice to avoid defining a variable until its use is required. In the CHIPS module, we have encountered a number of cases in which variables are defined early in a function, before they are needed. Often, this implies that their initial values must be adjusted before they are actually used. By delaying such variables' definitions, they can be correctly initialized and ready for immediate use. [L/M]

## 3 Class-specific observations

The following remarks encompass only classes and functions that we encountered during our review. It is not intended as a comprehensive list for the entirety of the CHIPS module. Moreover, given the extreme size of some of the classes and functions, these remarks may not be comprehensive even within the specific portions we call out below.

### 3.1 `G4QCaptureAtRest`

Member function `AtRestDoIt()` is another rather long function, comprising lines 135-899.

It ensures, but in a rather strange way, that the `CHIPSWorld` singleton is created.

### 3.2 `G4QCHIPSWorld`

This class is designed as a singleton. (Actually, the class provides not one but two singletons: one of type `G4QCHIPSWorld`, the other of type `G4QParticleVector`. Code inspection suggests that initially the class provided only the former, and that it subsequently accreted the latter.)

- While the copy constructor and copy assignment operator are correctly declared and not defined, marginal annotations incorrectly proclaim they take their arguments "by value."
- The class unnecessarily declares equality and inequality operators, as well as a so-called copy "by pointer" function. None of these is needed; copying is counter to the design of any singleton, and if there is only one, there can be no other to compare against.
- There seems to be no coherent policy regarding the use of inlining. Several functions could be trivially inlined for improved performance (e.g., the default constructor at line 45) but are not.
- It is unusual in a singleton class to have the constructor `protected` rather than `private`. Such declaration seems to argue for inheritance, but one can't inherit from any singleton class.
- `GetParticles` has an unusual semantic for an accessor function: it has a side effect of ensuring that a specified minimum number of particles are available in the `vector` it returns. This side effect leads to unusual calls (e.g. line 424 in `G4StringChipsParticleLevelInterface`) just for the side effect and not for any access. (Indeed, the result returned by the accessor call is discarded.)

- In the header of this class, the member function `GetQPEntries()` seems suboptimally named; it currently suggests that it returns entries, yet returns only a container's size. Moreover, its return type, a signed type, requires that a conversion be performed, as such sizes are generally unsigned integral types.

## 3.3 `G4ChiralInvariantPhaseSpace`

In member function `ApplyYourself()`, we note suboptimally structured code. For example, the loop at lines 135-171 has the following form:

```
for( ... ) {
  if( ... != ... ) {
    delete ...
    continue;
  }
  ...
  delete ...
}
```

Since the two `delete`'s in the above code are identical, the code fragment can be transformed into:

```
for( ... ) {
  if( ... == ... ) {
    ...
  }
  delete ...
}
```

This restructured code is cleaner, shorter, and easier to understand and maintain.

## 3.4 `G4QElasticCrossSection`

- We note that member function `GetTabValues()` is the function most frequently called during our testing.

- This class has a number of members that are of type `std::vector<G4double*>`. Containers holding pointers to simple `double`s (rather than holding the `double`s themselves) seems both unnecessary and expensive.

- Although this class (as well as other `G4Q*CrossSection` classes) are structured as a classic singleton, it is unclear why this is needed, since all its data members are static, the class behaves much like a singleton without the complexity of calls to any instance function.

## 3.5 `G4QEnvironment`

**Re the equality/inequality operators.** The operators `==` and `!=` are coded as tests for identity, not equality. This is unintuitive and may suggest some hidden dependencies involving the use of pointers.

**Re member function `CopyAndDeleteHadronVector`:**

- This function poorly named as it does not necessarily copy the entire vector it is given (although it does dispose of its contents).

- We note that this function makes fresh copies of the selected items rather than simply taking possession of their pointers and replacing the original pointers with null pointer values 0 (since the originals are about to go away anyway). This leads to poorer-than-necessary performance.
- The opening test `if(nHadrons)` seems unnecessary.

**Re member function `Fragment`:**

- The purpose of lines 4120-4130 seems opaque to understand:

```
4120 G4int tmpS=intQHadrons.size();
4121 if(tmpS)
4122 {
4123   tmpS=theFragments->size();
4124   intQHadrons.resize(tmpS+intQHadrons.size()); // Resize intQHadrons
4125   copy(theFragments->begin(), theFragments->end(),
             intQHadrons.end()-tmpS);
4126   tmpS=intQHadrons.size();
4127   theFragments->resize(tmpS); // Resize theFragments
4128   copy(intQHadrons.begin(), intQHadrons.end(),
             theFragments->begin());
4129   intQHadrons.clear();
4130 }
```

  It seems that the same effect is achievable via the following two lines of code, and with considerably improved performance since it uses no extra variable and makes fewer copies:

```
theFragments.insert( theFragments.begin()
                   , intQHadrons.begin(), intQHadrons.end()
                   );
intQHadrons.clear();
```

- This function uses local pointers (*e.g.*, variable `reQuasmons` at line 4014 and variable `reQHadrons` at line 4024) in conjunction with dynamically-allocated pointees where simple local variables of the pointee types would suffice. This seems to introduce unnecessary overhead and complexity in this frequently-called function.

**Re member function `FSInteraction`:**

- This function is almost 2900 code lines in size. There are copious debugging statements, contributing to the bulk, but it's hard to know whether there is lots of debugging because the function is big, or it's big because there is lots of debugging.
- The indentation is inconsistent, making it difficult to understand and follow the heavily-nested logic.

**Re member function `HadronizeQEnvironment`:**

- Line 1351 exemplifies the use of a pointer as an alias (handle) to an existing pointee; it has no responsibility for its pointee's lifetime.
- In lines 1481-1484 and again at 3582-3585 we see a sequence in which the last two items are removed from a `vector`, and in the next line the 2nd-last item is restored to the identical position from which it was just removed. The need for this behavior seems puzzling.

## 3.6 `G4QHadron`

We note that the copy constructor (beginning at line 147) fails to copy two of the class' data members; it also fails to make any explicit use of the initializer list. Moreover, this constructor injects known (constant) values for two of the data members of the resulting putative copy.

## 3.7 `G4QNucleus`

**Re incomplete constructors:** At least some constructors fail to initialize all the data members. This suggests that those data members are not really part of the state of such an object.

**Re copy construction:** A copy constructor is commented out, leaving the compiler to generate one. While this is fine, another "copy-like" constructor (taking a pointer) is present but does not provide a faithful copy. Instead, it injects a known constant value (namely `-1`) into a data member of the purported copy. (The commented-out copy constructor also had this "faithless copy" behavior. If this is necessary for the correctness of the algorithms involved, we conjecture that, that commenting out the constructor may have subtly changed the behavior.)

**Re conversion constructors:** The class has several conversion constructors that are not marked as `explicit`.

**Re loop improvements:** Certain explicit loops seem to be candidates for replacement by calls to standard algorithms. For example, we have:

```
probVect[0]=mediRatio;
for(G4int i=1; i<256; i++) {probVect[i] = 0.;}
```

This could be more simply written:

```
probVect[0]=mediRatio;
std::uninitialized_fill( probVect+1, probVect+256, 0.0 );
```

Alternatively, a change of data structure, using an `std::vector<>` instead of an array, would lead to:

```
probVect(256, 0.0)       // in the initializer list
probVect[0]=mediRatio;   // in the body
```

## 3.8 `G4QParticleVector` and `DeleteQParticle`

It is unclear that these classes need at all to exist as independent entities. With suitable adjustments (*e.g.*, by making the container hold objects instead of pointers to those objects, or by making the container hold smart pointers instead of native pointers), these classes' *raison d'etre* goes away.

However, if it is necessary for them to exist, the following observations become relevant.

- This class inherits from `std::vector<>`, but (like most classes in the standard library) `std::vector<>` is not designed for inheritance.
- The `G4QParticleVector` destructor carefully deletes pointees in their order of creation, but the reverse order of creation is usually a more appropriate order since the canonical order of destruction is the reverse order of creation.

- `G4QCHIPSWorld::GetQWorld()` appears to be the only consumer of these classes. It is unclear why there is no `G4QParticleVector` constructor to enforce the 10-element minimum that this consumer seems to require, as this would lead to some consumer simplification.
- `DeleteQParticle` does not need to be a class. It could easily be replaced by a free function or even by a static member function (of `G4QParticleVector`), in either case of the form:

```
void deleteQparticle( G4QParticle * aN)
{ delete aN; }
```

## 3.9 G4QQPDGCode

**Re member function ConvertPDGToZNS.** This very frequently called function heavily uses literals in place of named constants.

**Re member function MakeQCode:**

- We note that the parameter of this very frequency-called function has type `int const &;` rather than the simpler type `int`, leading to a possible slight performance degradation due to the overhead associated with references.
- The internal structure of this code is heavily based on a sequence of tests probing for specific values for a number of variables, instead of on straightforward `switch` statements. See for example lines 408-413, 417-421, 430-441, *etc.*

## 3.10 G4QStringChipsParticleLevelInterface

**Re the default constructor.** This function is missing an explicit initializer list. Instead, it relies on compiler-generated defaults to initialize both its base class subobject and its data member `theModel`.

**Re the member function Propagate():**

- This function seems unnecessarily bulky. As but a small example, lines 108,114,115 can be combined and rewritten as a single line at 115:

```
return new G4ReactionProductVector(1, theFastSec);
```

- Postincrement is consistently used, *e.g.*, at lines 134, 141, and 171, and in all loops.
- In the loop at line 171, the use of `unsigned int` as the counting type is suspect as there is no guarantee that this is the type of the result returned by the call to `size()` in the loop predicate. If it is not, each iteration will involve an unnecessary type conversion. (The type returned by `size()` is `G4ReactionProductVector::size_type`.)
- In the same loop, the counting variable `secondary`, although of unsigned type, is initialized with a signed zero (`0`) rather than an unsigned zero (`0u`), thus incurring the cost of a conversion (either at compile time or at run time).
- Also in the same loop, `theSecondaries` is an `std::vector<>` whose size seems not to change from one iteration to the next, yet `size()` is re-evaluated during each iteration.
- The loop predicate (in the same loop) is phrased in terms of the `<` operator rather than the `!=` operator.

- Constants (see lines 532-548) are in the form of literals (*e.g.*, 90000000) rather than as named constants defined in a single place.
- At line 512, the `if` guards a loop with essentially the identical condition and is thus redundant.
- Pointers are used extensively, but lack any consistent and clear ownership policy of their respective pointees.

### 3.11 G4Quasmon

**Re data member theWorld.** This data member seems to exist solely to cache a pointer to the single instance of `G4QCHIPSWorld`, a class documented and implemented as a singleton type. Such caching seems unusual, and is unnecessary for singletons, as it unnecessarily consumes extra space and time. A singleton's `instance()` function (as it is traditionally named) is typically inlined, and therefore has no overhead beyond that of copying the (pointer or reference to) the singleton object.

**Re member function HadronizeQuasmon.** This function is comprised of lines 219-3591. Without first breaking it into individually manageable pieces, this function may well be difficult to maintain.

**Re static member functions SetTemper, SetSOverU, SetEtaSup, SetParameters, *etc.*** It is unclear why these small functions are not implemented `inline`.

**Re member function RandomPoisson.** It is unclear how this function differs from `G4Poisson`.

**Re member function CalculateHadronizationProbabilities:**

1. This function consists of roughtly 600 lines, almost all of which is a single loop.
2. Within this function, constants seem used inconsistently. For example, we have `NUCPDG` defined, yet its value is also used as a literal in several places.

## 4 An experiment in code improvement

In order to assess the feasibility of adjusting the code constituting the CHIPS module, we made a limited modification of the code. In particular, we decided to change a single `typedef` and follow the consequences of that change. As shown in the following code fragment, we replaced a `vector` of pointers with a `vector` of the pointees instead. Note that, as an immediate consequence of the update, we no longer needed the `DeleteQuasmon` function object type:

```
// G4QuasmonVector, lines 41-2:
typedef std::vector<G4Quasmon *> G4QuasmonVector;
struct DeleteQuasmon{ void operator()(G4Quasmon *aN){delete aN;} };

//transformed to contain pointees instead of pointers:
typedef std::vector<G4Quasmon> G4QuasmonVector;
```

Almost all of the consequent changes (the ripple effects of the above-described initial change) followed one of two patterns, representative examples of which are shown below.

In the first pattern, we avoid the construction of a heap-based object. Instead, we construct an equivalent temporary object on the stack, then append it directly to the `vector` in the same manner as previously used to append a pointer. The temporary is then implicitly destroyed, as it is no longer needed. This approach avoids any pointer use (both here as well as in all subsequent accesses), but does introduce an implied additional copy via the copy constructor.

```
// G4QEnvironment.cc, lines 444-5:
G4Quasmon* curQuasmon = new G4Quasmon(hQC, h4Mom);
theQuasmons.push_back(curQuasmon);

// transformed, avoiding the heap by introducing a copy instead:
theQuasmons.push_back( G4Quasmon(hQC, h4Mom) );
```

The second pattern is similar, but has a technical difference. The original code, given a pointer to an object, constructed on the heap a copy of the pointee object, then appended to a `vector` the pointer to the copy. Again, the transformed code avoids any use of the heap; it directly appends (via an implied copy) the original object to the `vector`:

```
// G4QEnvironment.cc, lines 4020-1:
G4Quasmon* curQ     = new G4Quasmon(theQuasmons[iq]);
reQuasmons->push_back(curQ);

// transformed, avoiding the heap with no change in copying:
reQuasmons->push_back( theQuasmons[iq] );
```

We note that both transformations resulted in different constructors being called. The first inserted a call to a copy constructor, the second replaced a call to a “copy constructor by pointer” with a call to the traditional copy constructor. Both transformations are designed to avoid any use of dynamic memory management.

We selected the above set of transformations for several reasons: We believed that it was representative of a class of transformations that would yield performance improvements. Moreover, the functions containing the affected code had ranked high on our list of frequently-invoked functions. In addition, we believed the transformed code would be more amenable to future maintenance, should it become necessary or desireable. Finally, we expected the transformations to be limited in scope, and in fact they predominantly affected the functions in only a very few classes.

The transformations were accomplished in roughly four hours of editing. We expected that the transformed code would, after recompiling the modified CHIPS module and relinking it with the adapted novice example N04, we would obtain the same results as the unmodified code, with somewhat improved performance.

Along the way, we uncovered an additional opportunity for improvement. Specifically, we discovered pointers used for aliasing purposes, namely as simple handles to data. Such handles have no responsibility for ownership, and are specifically not used to terminate their pointee’s lifetime. This use, and recommendations for its improvement via the transformation shown below, are now incorporated among this document’s recommendations.

```
// G4QEnvironment.cc, line 1351:
G4Quasmon*      pQ = theQuasmons[is];

// transformed, using a reference type to connote the aliasing:
G4Quasmon&      pQ = theQuasmons[is];
```

We were in fact able to reproduce the original output, but could not detect a measurable difference in timing using the small novice example (only three events) at our disposal. However, using the CHIPS part of `test35` on 500,000 events, we observed a performance improvement on the order of 0.5%.

## 5 Possible sequence for recommended modifications

The following sequence and accompanying time estimates (in brackets) are based on the recommendations found in the previous sections of this paper and on our assessment (also documented above) of their relative importance and potential difficulty. The sequence additionally attempts to group related modifications into proximity with each other. Finally, the sequence includes, in its earlier items, tasks that are prerequisite to items later in the list.

1. Provide for testing, perhaps making the test programs reusable as examples for others to follow [days per use case]
2. Improve the code documentation on an ongoing basis throughout this modification process, along with general code appearance and use of debugging statements [absorbed by other items]
3. Repair all special member functions [days to weeks per class]:
   - Make maximal use of constructors' initializer lists
   - Ensure that constructors and assignment operators initialize all members and bases
   - Ensure that copy constructors and assignment operators make accurate copies, providing current "partial copy" and/or "copy with reset" functionality, if needed, via differently-named functions
   - Discard the unnecessary "copy constructor by pointer" functions
   - Mark all conversion constructors as `explicit`
   - Revise equality and inequality operators to provide standard semantics rather than identity testing; omit these entirely if unused
   - Revise copy assignment implementation as recommended above, providing missing `swap` functions as needed
4. Minimize pointer use [days to weeks per class]:
   - Replace each type of the form `vector<T*>` with a type of the form `vector<T>`
   - Change the type of alias variables from `T*` to `T&` or, if possible, to `T const&`
   - Replace each remaining local variable created via a definition of the form `T* t = new T(...);` with a local variable created via `T t(...);` instead
   - Indicate optional arguments via a type (such as `boost::optional<>`) designed for this purpose
   - Replace any remaining variables of pointer type with equivalent variables of a suitable smart pointer type (such as the forthcoming `std::unique_ptr<>` or `std::shared_ptr<>`)
5. Improve looping control [weeks]:
   - Remove `if`'s that guard `for`- and `while`-loops
   - Replace post-increment operators with pre-increment operators wherever the returned value is not used in loop reinitialization
   - Avoid unnecessary recalculations in loop predicates

- Employ `operator !=` wherever possible in loop predicates
- Replace loops that copy with an appropriate call to a standard library function
- Determine and consistently apply a policy governing choice of types for loops controlled by a counting variable; prefer to control loops via iterators instead of counters wherever possible

6. Improve use of constants [weeks to months]:
   - Replace non-trivial literals with appropriately named constants
   - Apply `const` to all variables intended never to be changed
7. Restructure code [weeks to months]:
   - Avoid excessively bulky functions
   - Avoid declaring variables before they are needed
   - Take advantage of other opportunities to streamline code
8. Investigate inheritance relationships [weeks]:
   - If not moot due to above modifications, avoid use of inheritance from most standard library facilities
   - Consider `private` inheritance
9. Finally, consider restructuring the module based on its current, evolved, functionality [weeks to months for design; months to implement]

## 6 Acknowledgments

We appreciate advice and assistance received from Mark Fischler, Jim Kowalkowski, Marc Paterno, and Julia Yarba.