C++11 Guidelines

I. Hrivnacova, IPN Orsay

20th Geant4 Collaboration Meeting,
30 September 2015, Fermilab

Introduction

“C++11 feels like a new language.”

The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever.

…

In other words, I'm still an optimist.

– Bjarne Stroustrup

C++11 As A Revolution

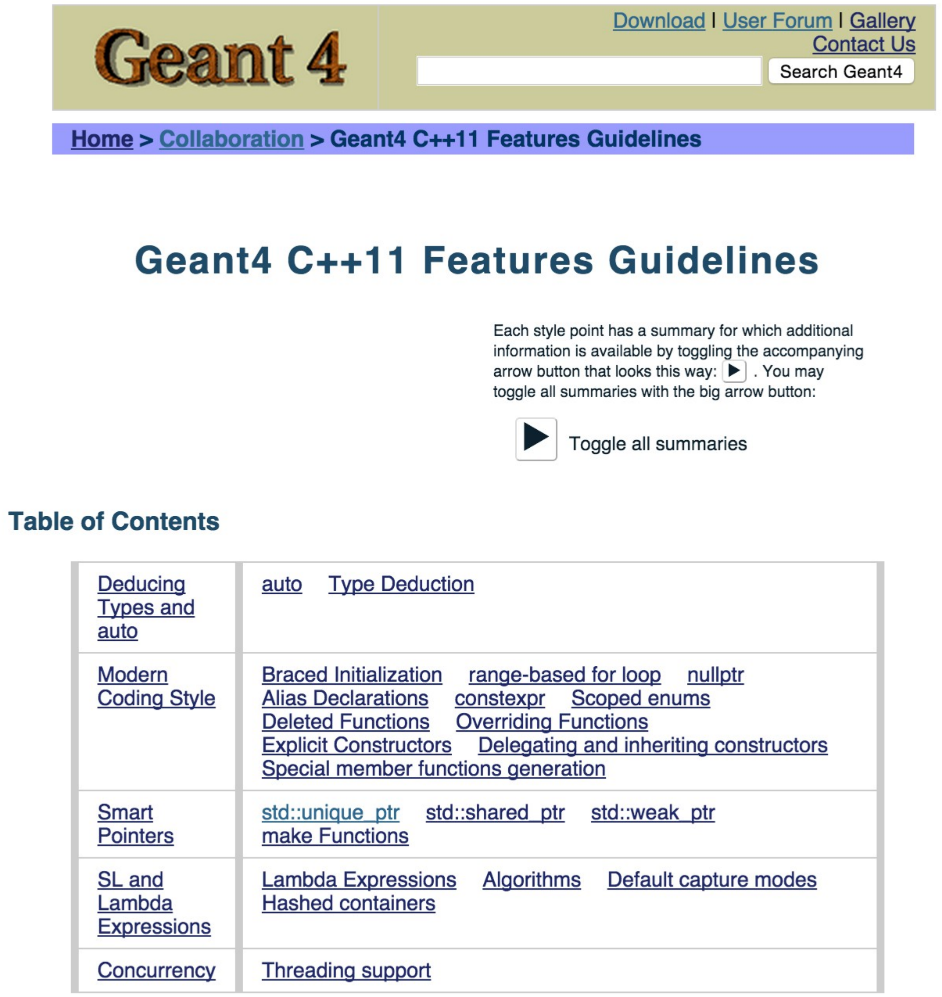
- Lambda expressions - let you define functions locally, at the place of the call
- Automatic type deduction - you can declare objects without specifying their types
- Rvalue references - can bind to “rvalues”, e.g. temporary objects and literals.
- Smart pointers – no delete
- C++ Standard library - new container classes, new algorithm library and several new libraries for regular expressions, tuples, …
- Threading library – thread class

About This Mini-Course

- There is a lot to learn about C++11
- The Geant4 C++11 guidelines document and this mini-course as a starting point
- Not all important features are covered in the document
  - Move semantics, New features in C++ Standard library or Threading library
  - Due to lack of time and/or lack of experience (Standard Library, Move semantics) or because they will be applied by the core developers and are not supposed to be used directly by Geant4 developers (Threading)
- More attention is given to Move semantics in this mini-course than in the document

C++11 Guidelines Document

- Compiled from the following sources:
  - Effective Modern C++ by Scot Meyers (O'Reilly). Copyright 2015 Scot Meyers. 978-1-491-90399-9.
  - ALICE O² C++ Style Guide.
  - cplusplus.com
  - Stack Overflow
  - *It is using style sheets of C++ Google Style guide, Revision 3.274 (link) under the CC-By 3.0 License further modified by ALICE O² project*



Home > Collaboration > Geant4 C++11 Features Guidelines

Geant4 C++11 Features Guidelines

Each style point has a summary for which additional information is available by toggling the accompanying arrow button that looks this way: ▶ . You may toggle all summaries with the big arrow button:

▶ Toggle all summaries

Table of Contents

Deducing Types and auto	auto Type Deduction
Modern Coding Style	Braced Initialization range-based for loop nullptr Alias Declarations constexpr Scoped enums Deleted Functions Overriding Functions Explicit Constructors Delegating and inheriting constructors Special member functions generation
Smart Pointers	std::unique_ptr std::shared_ptr std::weak_ptr make Functions
SL and Lambda Expressions	Lambda Expressions Algorithms Default capture modes Hashed containers
Concurrency	Threading support

http://geant4.cern.ch/collaboration/c++11_guide.shtml

Guidelines

- The guidelines are the guidelines, not rules.
- With each guideline we give the rationale behind this guideline
- The guidelines grouping was mostly inspired by Meyer's book:
  - Deducing types and autos
  - Modern coding style
  - Smart pointers
  - Standard Library and Lambda Expressions
  - Concurrency

Legend To The Slides

```
A simple example of
C++11 code discussed.
```

- *The guideline text.*

```
// code which is error-prone or wrong
// and which can be avoided using C++11 features
```

```
// code using C++98 features
// which can be improved with use of C++11 features
```

```
// code using C++11 features
```

Deducing Types and `auto`
Braced Initialization

In addition to C++98 type deduction for function templates, C++11 adds two more: auto and decltype(*).

(*) Not explained in this mini-course

auto

```
auto x = 0;
```

- *Prefer `auto` to explicit type declarations.*

- Auto variables must be always initialized

```
int x1;
```

```
auto x1;
```

```
auto x3 = 0;
```

- Compiles but x1 may be used further without being initialized
- Produce compilation error
- This code is ok

auto (2)

- Auto variables are immune to type mismatches

```
std::unordered_map<std::string, int> m;
for (const std::pair<std::string, int>& p : m) {
  ...   // do something with p
}
```

  - The p type in the loop does not match the map m element type, which is **std::pair<const std::string, int>** (note the const)

```
std::unordered_map<std::string, int> m;
for (const auto& p : m) {
  ...   // do something with p
}
```

Braced Initialization

```
std::vector<int> myVector{0, 1, 2};
```

- *Distinguish between* `()` *and* `{}` *when creating objects.*

- Prevents from narrowing conversion

```
double x, y, z;
int sum1(x  + y + z);
int sum2 = x  + y + z;
```

  - Conversion double → int

```
double x, y, z;
int sum1{ x  + y + z };
```

  - Compiler ERROR

- Prevents from "most vexing parse"

```
Widget w1(10);
```

  - Call Widget constructor with argument 10

```
Widget w2();
```

  - Declares **a function named "w2"** that returns a Widget

```
Widget w3{};
```

  - Call Widget constructor with no arguments

Braced Initialization
std::initializer_list

```
std::initializer_list<T> il;
```

- In constructor calls, parentheses and braces have the same meaning as long as std_initializer_list parameters are not involved
- Otherwise the parameters are matched to std::initializer_list parameters if at all possible

```
class Widget {
public:
 Widget(int i, bool b);   #1
 Widget(int i, double d);#2
};


Widget w1(10, true); // #1
Widget w2{10, true}; // #1
Widget w3(10, 5.0);  // #2
Widget w2{10, 5.0};  // #2
```

```
class Widget {
public:
 Widget(int i, bool b);    #1
 Widget(int i, double d); #2
 Widget(                   #3
  std::initializer_list<double> il);
};

Widget w1(10, true); // #1
Widget w2{10, true}; // #3 !!
Widget w3(10, 5.0);  // #2
Widget w2{10, 5.0};  // #3 !!
```

Braced Initialization
std::initializer_list (2)

- Be careful when creating a std::vector of a numeric type with two arguments

```
std::vector<int> v1(10, 20);
```

- Creates 10-element std::vector, all elements have value of 20

```
std::vector<int> v2{10, 20};
```

- Creates 2-element std::vector, all element values are of 10 and 20

- Never assign a braced-init-list to an auto local variable

Type deduction

- *auto-typed variables can be subject of pitfalls.*

- Some of auto's deduction type results, while conforming to the prescribed algorithms, may be different from the programmer expectations.

```
auto x1 = 27;
auto x2(27);
```

- x1, x2 type is **int**

```
auto x3 = { 27 };
auto x4{ 27 };
```

- x3, x4 type is **std::initializer_list<int>**

Modern Coding Style

Range-based for loop, nullptr, Alias Declarations, constexpr, Scoped enums, Deleted Functions, Overriding Functions, Explicit Constructors, Delegating and inheriting constructors, Special member functions generation

Range-based for loop

```
for (auto i : {0, 1, 2}) { .. }
```

- *Prefer range-based for loop when iterating over all elements in a container or a braced initializer list.*

- Three use cases of auto-declaration:

```
std::vector<int> v = {0, 1, 2, 3, 4, 5};
```

```
for (auto i : v)
{
   ..
}
```

```
for (const auto& i : v)
{
   ..
}
```

```
for (auto&& i : v)
{
   ..
}
```

- Access by value, i type is **int**
- Access by const reference, i type is **const int&**
- Access by reference, i type type is **int&**

nullptr

```
Widget* widget = nullptr;
```

- *Prefer* `nullptr` *to* `0` *and* `NULL.`

- Using nullptr instead of 0 and NULL avoids overload resolution surprises, because nullptr can't be viewed as an integral type.

Alias Declarations

```
using MyType = std::vector<int>;
```

- *Prefer alias declarations to typedefs.*

- typedefs do not support templatizations, but alias declarations do.
- The syntax with alias declarations is easier to understand in some constructs (e.g. function pointers).

```
// C++98
typedef std::map<double, double> MyMap;
```

```
// C++11
using MyMap = std::map<double, double>;
```

constexpr

```
constexpr auto size = 10;
```

▶ *Use constexpr whenever possible.*

Not supported on “vc12” !

- Some variables can be declared constexpr to indicate the variables are true constants, i.e. they are initialized with values known during compilation.

```
constexpr auto arraySize = 10;
std::array<int, arraySize> data;
```

- Some functions and constructors can be declared constexpr which enables them to be used in defining a constexpr variable.

```
constexpr int getDefaultArraySize (int multiplier)  {
  return 10 * multiplier;
}
std::array<int, getDefaultArraySize(3)> data;
```

Scoped enums

```
enum class Color { black, white, red };
```

- *Prefer scoped* `enums` *to unscoped* `enums`.

- C++11 enums declared via enum class don't leak names:

```
enum Color98 {
  black, white, red };

Color98 c = white;
```

  - Ok ! enumerator "white" is in this scope

```
enum class Color11 {
  black, white, red };

Color11 c = white; // ERROR
Color11 c = Color11::white;
```

  - Scope must be provided

- Scoped enums prevent from implicit type conversions, they convert to other types only with a cast.

- Scoped enums may always be forward-declared

```
enum Color98;   // ERROR
enum Color98: int; // Ok
```

```
enum class Color11;
```

Deleted Functions

```
Widget(const Widget&) = delete;
```

- *Prefer deleted functions to private undefined ones.*

```
class Widget {
...
private:
 Widget(const Widget&);
 Widget& operator=(const Widget&);
};
```

```
class Widget {
public:
 Widget(const Widget&) = delete;
 Widget& operator=(const Widget&)= delete;
};
```

- Any function may be deleted, including non-member functions and template instantiations. They can be use to invalidate some undesired types in overloading.

Overriding Functions

```
virtual void f() override;
```

- *Declare overriding functions* `override` *or* `final`*.*

```
class Base {
public:
 ...
 virtual void mf1() const;
};
```

```
class Derived : public Base {
public:
  ...
  virtual void mf1();
};
```

```
class Derived : public Base {
public:
  ...
  virtual void mf1() override;
};
```

- Will compile with WARNING
- Will NOT compile - ERROR

- The `final` keyword tells the compiler that subclasses may not override the virtual function anymore.

Explicit Constructors

```
explicit Widget(int number);
```

- *Declare constructors with one argument* `explicit`, *except for copy constructors and constructors with* `std::initializer_list`.

```
void f(const Widget& widget) { ... }
```

```
class Widget {
public:
 Widget(int number);
};
```

```
class Widget {
public:
 explicit Widget(int number);
};
```

```
f(5);
```

- The Widget(5) constructor will be called and passed in function f
- Compiler error: no matching function for call to 'f'
- Declaring a constructor `explicit` prevents an implicit conversion.

Delegating and Inheriting Constructors

- *Use delegating and inheriting constructors when they reduce code duplication.*

- Example of a delegating constructor:

```
class Widget {
 public:
  Widget::Widget(const string& name) : mName(name) { }
  Widget::Widget() : Widget("example") { }
  ...
};
```

- Example of an inheriting constructor:

```
class Base {
 public:
  Base();
  explicit Base(int number);
  ...
};
```

```
class Derived : public Base {
  public:
    using Base::Base;
};
```

Move Semantics

Passing objects by value, Lvalue, Rvalue, &&, Special member functions

This section was also inspired by https://mbevin.wordpress.com/2012/11/20/move-semantics/

Passing Large Objects

- With C++98 - large objects are returned from functions by reference or by pointer to avoid expense copying

```
vector<int>* makeBigVector1()
{..}
...
vector<int>* v1 = makeBigVector1();
delete v1;
```

```
void makeBigVector2(vector<int>& out)
{..}
..
vector<int> v2;
makeBigVector2(v2);
```

- With C++11 move semantics they can be simply return by value

```
vector<int> makeBigVector()
{..}
...
auto v = makeBigVector();
```

- All STL collection classes have been extended to support move semantics
- The content of the temporary created vector in the function body is moved in 'v' and not copied

Rvalue, lvalue, &&

- New concepts of rvalues and lvalues in C++11
- an **lvalue** is an expression whose address can be taken, a locator value. Anything you can make assignments to is an lvalue
- an **rvalue** is an unnamed value that exists only during the evaluation of an expression
- the **&&** operator is new in C++11, and is like the reference operator (&), but whereas the & operator can only be used on lvalues, the && operator can only be used on rvalues.

- `int x = 1  - x` is lvalue, 1 is rvalue (l=left, r=right)

Rvalue, lvalue, && - Example

```
class A {
public:
 static A inst;
 static A& getInst()    { return inst; }//#1
 static A  getInstCopy(){ return inst; }//#2
};
```

- #1 is returning a reference to a static variable, hence it's returning an **lvalue**
- #2 is returning a temporary copy of instance, hence it's returning an **rvalue**

```
A&  inst1  = A::getInst();
A&& inst2 = A::getInst();
A::getInst() = A();
A   inst3 = A::getInstCopy();
A&  inst4 = A::getInstCopy();
A&& inst5 = A::getInstCopy();
```

1. ok - we've fetched a reference to the static instance variable
2. ERROR - can't assign a reference to an rvalue reference

- getInst() is an lvalue reference, we assign a new value to it

3. ok - we've fetched a copy of the instance
4. ERROR- can't assign a reference to a temporary (an rvalue)
5. ok - we've assigned an rvalue reference to the temporary copy that was made of the instance

When Moving Is Possible

- When passing an object to a function (or returning it from a function), it's possible to do move (rather than a copy) if:
  - the object is an **rvalue**
  - the object class defines t**he special member move function**
- When move occurs, data is removed from the old object and placed into a new object; the compiler can only do a move if
  - The old object is temporary
  - When **std::move** is called explicitly on an object

```
string s1("abcd");
cout << "s1: " << s1 << endl;
std::string s2(std::move(s1));
cout << "s1: " << s1 << endl;
cout << "s2: " << s2 << endl;
```

Will produce output:

```
s1: abcd
s1:
s2: abcd
```

Special Member Functions

- The special member functions are those compilers may generate on their own:
  - default constructor, destructor, copy operations
  - move operations (C++11 only).

```
class Widget {
public:
  Widget(const Widget&& rhs);
  Widget& operator=(const Widget&& rhs);
};
```

- The behavior of a class which relies on generating all special member functions can be accidentally changed by adding one of these functions, e.g. a destructor for logging functionality.
- That's why it's important to understand the C++11 rules governing this automatic generation.

Special Member Functions Generation

- *Understand special member functions generation.*

- Move operations are generated only for classes lacking explicitly declared moved operations, copy operations, and a destructor.
- The copy constructor is generated only for classes lacking an explicitly declared copy constructor, and it's deleted if a move operation is declared.
- The copy assignment operator is generated only for classes lacking an explicitly declared copy assignment operator, and it's deleted if a move operation is declared. Generation of the copy operation in classes with an explicitly declared destructor is deprecated.
- Member function templates never suppress generation of special member function.

Special Member Functions Generation (2)

- If e.g. a destructor for logging functionality is added to a class with no special member functio
- This will cause that only copy operations are generated and then performed instead of moving operation what can make them significantly slower.
  - When the behavior of compiler-generated functions is correct, you can declare this explicitly using **= default** keyword and make their existence independent from the implicit generation rules:

```
class Widget {
public:
  ~Widget();

  Widget(const Widget&& rhs) = default;
  Widget& operator=(const Widget&& rhs) = default;
};
```

Smart Pointers

Std::unique_ptr, std::shared_ptr, srd::weak_ptr, make functions

Smart pointers are objects that act like pointers, but automate ownership. They are extremely useful for preventing memory leaks. They also formalize and document the ownership of dynamically allocated memory.

std::unique_ptr

- *Use* `std::unique_ptr` *for exclusive-ownership resource management.*

- Small, fast, move-only

```
#include <memory>
{
  std::unique_ptr<int> uptr(new int(42));

  std::cout << uptr.get() << std::endl; // print a pointer value
  std::cout << *uptr      << std::endl; // print 42
}
// uptr is automatically freed here};
```

std::unique_ptr (2)

- The same object cannot be pointed by two unique pointers:

```
{
  std::unique_ptr<int> first(new int(1));
  std::unique_ptr<int> second = first;
}
```

  - Compiler error: “call to implicitly-deleted copy constructor of 'unique_ptr<int>' “

- Can be converted in `std::shared_ptr`:

```
{
  std::unique_ptr<int> uptr(new int(42));
  std::shared_ptr<int> sptr(std::move(uptr));
}
```

  - Note that after the move the unique pointer does not point to the int object anymore: `uptr.get()` will return `0x0`

std::shared_ptr

- *Use* `std::shared_ptr` *for shared-ownership resource management.*

- Garbage collection for the shared lifetime management of arbitrary resources
- Typically twice big as std::unique_ptr, overhead for control blocks, and requiring atomic reference count manipulations.

```
{
  std::shared_ptr<int> sh1(new int);
  std::cout << sh1.use_count() << std::endl;  // prints 1

  std::shared_ptr<int> sh2(sh1);
  std::cout << sh1.use_count() << std::endl;  // prints 2
  std::cout << sh2.use_count() << std::endl;  // prints 2
}
```

std::shared_ptr (2)

- Avoid creating std::shared_ptr from variables of raw pointer type.

```
{
  auto pw = new Widget;
  std::shared_ptr<Widget> spw1(pw);
  std::shared_ptr<Widget> spw2(pw);
}
```

  - Two control blocks for the same object, *pw, are created, and so also reference counts, each of which will eventually become zero, **that will lead to an attempt to destroy *pw twice.**

- Correct code

```
{
  std::shared_ptr<Widget> spw1(new Widget);
  std::shared_ptr<Widget> spw2(spw1);
}
```

  - The Widget is created via spw1 and spw2 uses then the same control block as spw1

std::weak_ptr

- *Use* `std::weak_ptr` *for* `std::shared_ptr`*-like pointers that can dangle.*

- Potential use cases for std::weak_ptr include caching, observer lists, and the prevention of std::shared_ptr cycles.

make Functions

- *Prefer* `std::make_unique`*(*) and* `std::make_shared` *to direct use of new.*

  - Compared to direct use of new, make functions eliminate source code duplication, improve exception safety and (some) make code faster

```
{
  std::unique_ptr<Widget> upw(new Widget>);
  std::shared_ptr<Widget> spw(new Widget>);
}
```

  - Without make function (previously was in green, now in red)

```
{
  auto upw1(std::make_unique<Widget>());
  auto upw1(std::make_shared<Widget>());
}
```

  - With make function (Widget type is not duplicated)

make Functions(2)

- `std::make_unique` is only part of C++14.
  - Its implementation can be however easily added in C++11 based code. Both a simple version and a full-featured linked in the guidelines document.
  - Simple implementation is also available in G4AnalysisUtilities.hh
- There are situations where use of make functions is inappropriate
  - See more details in the guidelines document

Standard Library and Lambda Expressions

Lambda Expressions, Algorithms, (Default) Capture Modes, Hashed Containers

Lambda Expressions

```
[]() { }
```

Understand `lambda` *expressions.*

- A lambda function in C++

```
[]() { }
```

- `[]` is the capture list, `()` the argument list and `{}` the function body
  - The argument list is the same as in any other C++ function.
  - The function body contains the code that will be executed when the lambda is actually called.
  - The capture list defines what from the outside of the lambda should be available inside the function body and how.
    - We will see this on the next slides

Lambda Expressions (2)

- Simple examples for understanding lambda syntax
- Use lambda as a function:

```
auto func = [] () { std::cout << "Hello world" << std::endl; };
func();
```

- Use lambda as an expression:

```
int i = ([](int j) { return 5 + j; })(6);
std::cout << "i=" << i << std::endl;
```

  - Will print i=11

- We will see more meaningful use of lambda expression with SL algorithms

Lambda Expressions (3)

- Unlike an ordinary function, which can only access its parameters and local variables, a lambda expression can also access variables from the enclosing scope(s).

```
int x = 10;
int i = ([x](int j) { return 5 + j + x; })(6);
std::cout << "i=" << i << std::endl;
```

  - Will print i=21

- The way how the external variables will be used in lambda is defined in **the capture list**. It can be either:
  - a value: [x]
  - a reference [&x]
  - any variable currently in scope by reference [&]
  - same as previous, but by value [=]
  - You can mix any of the above in a comma separated list [x, &y]

Default Capture Modes

```
[&]() { }
[=]() { }
```

▶ *Avoid default capture modes in lambda expressions.*

- A lambda with the default by-reference (or by-value) capture mode can take by reference (or by value) any variable that is currently in scope:

```
[&]() { /* do something here*/ }  // by-reference capture
[=]() { /* do something here*/ }  // by-value capture
```

  - Default by-reference capture can lead to dangling references. A problem can arise if the variable's life time is shorter than the life-time of the lambda and lambda can be then used with a dangling reference.
  - Default by-value capture can lead to dangling pointers (especially `this`).

Algorithms

- *Prefer algorithm calls to handwritten loops.*

- C++ standard library algorithms (std::for_each, std::find_if or std::transform are very efficient and can be very handy.
  - But difficult with C++98, particularly if the functor you would like to apply is unique to the particular function.
  - C++11 lambdas allow to write cleaner and shorter code
- An example follows on the next slide

Algorithms (2)

```
#include <algorithm>
#include <vector>

namespace {
struct f {
  void operator()(int) {
  // do something
  }
};
}

void func98(std::vector<int>& v) {
  f f;
  std::for_each(v.begin(), v.end(), 
f);
}
```

- If you only use f once and in that specific place it seems overkill to be writing a whole class
- Using lambda makes this cleaner to read (it keeps everything in one place) and potentially simpler to maintain

```
void func11(std::vector<int>& v) {
  std::for_each(v.begin(),v.end(), [](int) {/* do something here*/});
}
```

Lambda Expressions
A Closure

- In the previous example

```
void func11(std::vector<int>& v) {
  std::for_each(v.begin(),v.end(), [](int) {/* do something here*/});
}
```

- The highlighted **expression** is the lambda
- A **closure** is the runtime object created by a lambda
  - Depending on the capture list, it holds copies of or references to captured data
  - In the call to std::for_each above, the closure is the object which is passed at runtime as the third argument.
- A **closure class** is the a class from which a closure is instantiated. This class is generated by the compiler for each labda.

Hashed Containers

```
<unordered_map>
...
```

- *Since C++11 the standard library provides unordered containers in headers* `<unordered_set>`, `<unordered_multiset>`, `<unordered_map>` *and* `<unordered_multimap>`

- **Unordered maps** are associative containers that store elements formed by the combination of a key value and a mapped value, and which allows for fast retrieval of individual elements based on their keys.
  - Faster than `map` containers to access individual elements by their key
  - Generally less efficient for range iteration through a subset of their elements.
  - Implement the direct access operator (operator[]) which allows for direct access of the mapped value using its key value as argument.
  - Appropriate use of hashed containers can improve performance.
  - A link to examples at cplusplus.com and stackoverflow.com is provided in the guidelines document.

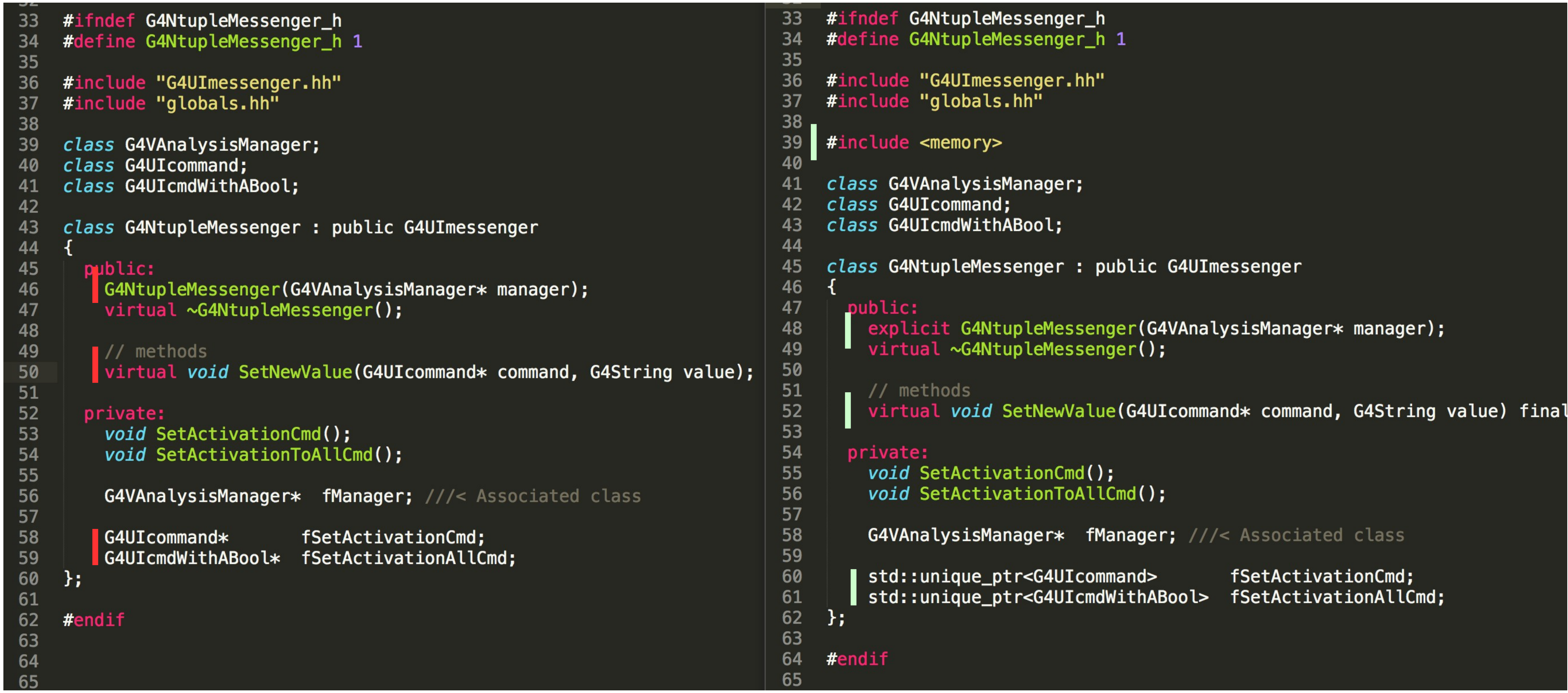
Threading Support

Threading Support

- *C++11 threading libraries should be used through the Geant4 interface and not directly.*

- C++11 provides support for multithreading in dedicated headers.
- Migration to C++11 will be done internally in Geant4 threading related core classes and definitions. In this way, they will be available through the Geant4 interface and should not be used directly.

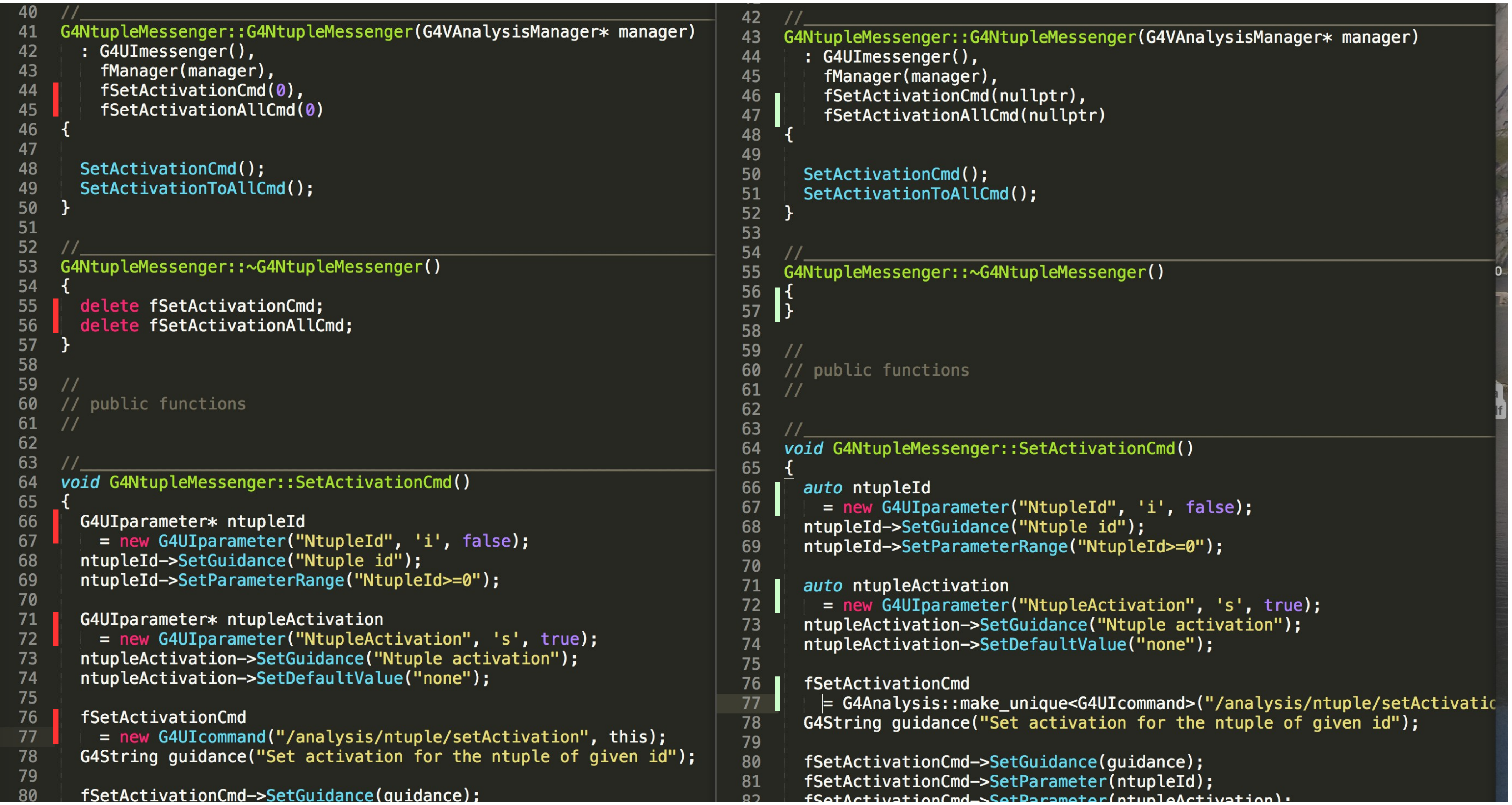
Example Of Code
in ref06 (C++98) – ref09 (C++11)



```cpp
#ifndef G4NtupleMessenger_h
#define G4NtupleMessenger_h 1

#include "G4UImessenger.hh"
#include "globals.hh"

class G4VAnalysisManager;
class G4UIcommand;
class G4UIcmdWithABool;

class G4NtupleMessenger : public G4UImessenger
{
  public:
    G4NtupleMessenger(G4VAnalysisManager* manager);
    virtual ~G4NtupleMessenger();

    // methods
    virtual void SetNewValue(G4UIcommand* command, G4String value);

  private:
    void SetActivationCmd();
    void SetActivationToAllCmd();

    G4VAnalysisManager*  fManager; ///< Associated class

    G4UIcommand*       fSetActivationCmd;
    G4UIcmdWithABool*  fSetActivationAllCmd;
};

#endif
```

```cpp
#ifndef G4NtupleMessenger_h
#define G4NtupleMessenger_h 1

#include "G4UImessenger.hh"
#include "globals.hh"

#include <memory>

class G4VAnalysisManager;
class G4UIcommand;
class G4UIcmdWithABool;

class G4NtupleMessenger : public G4UImessenger
{
  public:
    explicit G4NtupleMessenger(G4VAnalysisManager* manager);
    virtual ~G4NtupleMessenger();

    // methods
    virtual void SetNewValue(G4UIcommand* command, G4String value) final

  private:
    void SetActivationCmd();
    void SetActivationToAllCmd();

    G4VAnalysisManager*  fManager; ///< Associated class

    std::unique_ptr<G4UIcommand>       fSetActivationCmd;
    std::unique_ptr<G4UIcmdWithABool>  fSetActivationAllCmd;
};

#endif
```

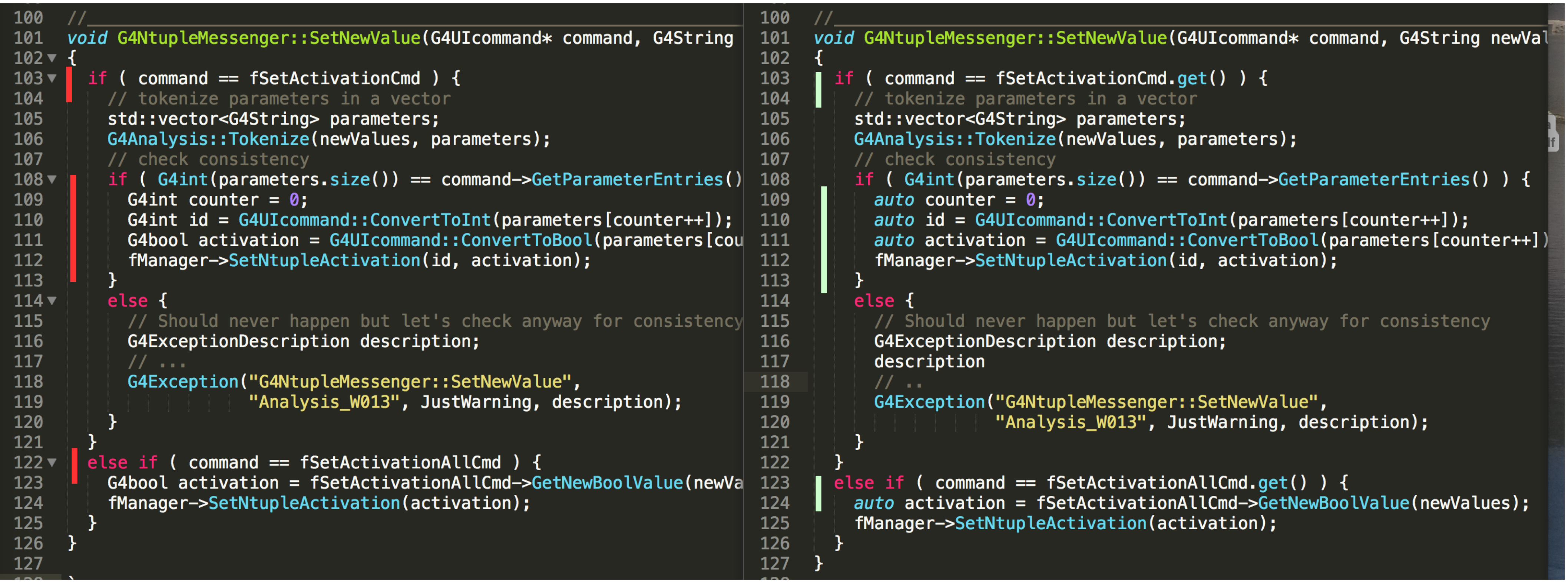
- Explicit constructor
- Overriding function declared final
- std::unique_ptr (requires including <memory>)



```
//
G4NtupleMessenger::G4NtupleMessenger(G4VAnalysisManager* manager)
  : G4UImessenger(),
    fManager(manager),
    fSetActivationCmd(0),
    fSetActivationAllCmd(0)
{

  SetActivationCmd();
  SetActivationToAllCmd();
}

//
G4NtupleMessenger::~G4NtupleMessenger()
{
  delete fSetActivationCmd;
  delete fSetActivationAllCmd;
}

//
// public functions
//

//
void G4NtupleMessenger::SetActivationCmd()
{
  G4UIparameter* ntupleId
    = new G4UIparameter("NtupleId", 'i', false);
  ntupleId->SetGuidance("Ntuple id");
  ntupleId->SetParameterRange("NtupleId>=0");

  G4UIparameter* ntupleActivation
    = new G4UIparameter("NtupleActivation", 's', true);
  ntupleActivation->SetGuidance("Ntuple activation");
  ntupleActivation->SetDefaultValue("none");

  fSetActivationCmd
    = new G4UIcommand("/analysis/ntuple/setActivation", this);
  G4String guidance("Set activation for the ntuple of given id");

  fSetActivationCmd->SetGuidance(guidance);
```

```
//
G4NtupleMessenger::G4NtupleMessenger(G4VAnalysisManager* manager)
  : G4UImessenger(),
    fManager(manager),
    fSetActivationCmd(nullptr),
    fSetActivationAllCmd(nullptr)
{

  SetActivationCmd();
  SetActivationToAllCmd();
}

//
G4NtupleMessenger::~G4NtupleMessenger()
{
}

//
// public functions
//

//
void G4NtupleMessenger::SetActivationCmd()
{
  auto ntupleId
    = new G4UIparameter("NtupleId", 'i', false);
  ntupleId->SetGuidance("Ntuple id");
  ntupleId->SetParameterRange("NtupleId>=0");

  auto ntupleActivation
    = new G4UIparameter("NtupleActivation", 's', true);
  ntupleActivation->SetGuidance("Ntuple activation");
  ntupleActivation->SetDefaultValue("none");

  fSetActivationCmd
    = G4Analysis::make_unique<G4UIcommand>("/analysis/ntuple/setActivatio
  G4String guidance("Set activation for the ntuple of given id");

  fSetActivationCmd->SetGuidance(guidance);
  fSetActivationCmd->SetParameter(ntupleId);
  fSetActivationCmd->SetParameter(ntupleActivation);
```

- Nullptr, no delete in destructor, auto
- The parameters ownership is handled by the framework, smart pointers should not be used in this case by the commands objects
- Using G4Analysis::make_unique (std::make_unique only in C++14)



```
//
void G4NtupleMessenger::SetNewValue(G4UIcommand* command, G4String
{
  if ( command == fSetActivationCmd ) {
    // tokenize parameters in a vector
    std::vector<G4String> parameters;
    G4Analysis::Tokenize(newValues, parameters);
    // check consistency
    if ( G4int(parameters.size()) == command->GetParameterEntries()
      G4int counter = 0;
      G4int id = G4UIcommand::ConvertToInt(parameters[counter++]);
      G4bool activation = G4UIcommand::ConvertToBool(parameters[cou
      fManager->SetNtupleActivation(id, activation);
    }
    else {
      // Should never happen but let's check anyway for consistency
      G4ExceptionDescription description;
      // ...
      G4Exception("G4NtupleMessenger::SetNewValue",
                  "Analysis_W013", JustWarning, description);
    }
  }
  else if ( command == fSetActivationAllCmd ) {
    G4bool activation = fSetActivationAllCmd->GetNewBoolValue(newVa
    fManager->SetNtupleActivation(activation);
  }
}
```

```
//
void G4NtupleMessenger::SetNewValue(G4UIcommand* command, G4String newVal
{
  if ( command == fSetActivationCmd.get() ) {
    // tokenize parameters in a vector
    std::vector<G4String> parameters;
    G4Analysis::Tokenize(newValues, parameters);
    // check consistency
    if ( G4int(parameters.size()) == command->GetParameterEntries() ) {
      auto counter = 0;
      auto id = G4UIcommand::ConvertToInt(parameters[counter++]);
      auto activation = G4UIcommand::ConvertToBool(parameters[counter++])
      fManager->SetNtupleActivation(id, activation);
    }
    else {
      // Should never happen but let's check anyway for consistency
      G4ExceptionDescription description;
      description
      // ..
      G4Exception("G4NtupleMessenger::SetNewValue",
                  "Analysis_W013", JustWarning, description);
    }
  }
  else if ( command == fSetActivationAllCmd.get() ) {
    auto activation = fSetActivationAllCmd->GetNewBoolValue(newValues);
    fManager->SetNtupleActivation(activation);
  }
}
```

- When the base class API is using raw pointers, we need to access them via smart pointers `get()` function
- More usage of `auto`

Further Reading

- Geant4 C++11 Guidelines document
  - http://geant4.cern.ch/collaboration/c++11_guide.shtml
- Effective Modern C++ by Scot Meyers (O'Reilly). Copyright 2015 Scot Meyers.
- C++ Core Guidelines by Bjarne Stroustrup, Herb Sutters:
  - https://github.com/isocpp/CppCoreGuidelines
- Or choose another one from the list of recommended books at https://isocpp.org/get-started

- And be prepared for C++14