Discussion of New C++11 Features

Internal Note for STAR Coding Guideline Committee

Monday, November 24, 2014

Thomas Ullrich

auto

Keyword **auto**: In C++11, if the compiler is able to determine the type of a variable from its initialization, you don't need to provide the type.

Example:

```
int x = 3;
auto y = x; // y is of type int
```

This is especially useful when working with templates and especially the STL.

Example:

```
map<string, string> addressBook;
addressBook["Joe"] = "joe@foo.com";
```

// add more people

Now you want to iterate over the elements of the **addressBook**. To do it, you need an iterator:

```
map<string, string>::iterator itr = address_book.begin();
```

In C++11 you now simply can write

```
auto itr = address_book.begin();
```

One can also use it this way:

```
auto i = 3;      // i is an int
auto x = 10.0;   // y is a double
auto l = 2LL;    // l is a long long
```

In the above examples auto does not really gain anything compared to

```
int i = 3;
double x = 3.2;
long long l = 2;
```

Also, the second version is certainly easier to read. Also, if a newbie programmer changes `10.0' to `10', x becomes an integral type and code depending on it to be a floating point type will fail.

Another pitfall can occur when using auto:

Bad example:

```
vector<int> numbers;
for (auto i = 0; i<numbers.size(); i++) {…}
```

This will create a warning since vector::size() returns an unsigned int and the condition compares integers of different types. The correct version is:

```
for (unsigned int i = 0; i<numbers.size(); i++) {…}
```

Now one certainly could use

```
for (auto i = 0UL; i<numbers.size(); i++) {…}
```

But this implies you have to specify the type anyway and thus defeats its purpose of **auto**.

Proposal for Guidelines

Use **`auto`** against verbosity, not consistency. The use of **`auto`** is useful in many cases where it increases consistency as for example in:

```
auto iter = someContainer.begin();
auto result = func();
```

In cases where the rhs expression is an integer or floating point literal the use of **`auto`** is strongly discouraged.

Use

```
int i = 0;
unsigned int k = 3;
long j = 5;
double x = 3.2;
```

instead of

```
auto int = 0;
auto k = 3U;
auto j = 5L;
auto x = 3.2;
```

Non-member begin() and end()

The non-member **`begin()`** and **`end()`** functions are a new addition to the standard library, promoting uniformity, consistency and enabling more generic programming. They work with all STL containers, but more than that they are overloadable, so they can be extended to work with any type. Overloads for C-like arrays are also provided.

To adopt any custom container all one must do is create your own iterator that supports *, prefix increment (++itr) and != .

Example:

```
int arr[] = {1,2,3};
for(auto iter=begin(arr); iter!=end(arr); iter++) {
    cout << *iter << endl;
}
```

For container classes that provide **`container::begin()`** and **`container::end()`** (such as **`vector<>`**) the non-member version can of course be used as well.
That is:

```
vector<int> vec;
auto iter1 = begin(vec);
auto iter2 = vec.begin();
```

`iter1` and **`iter2`** are identical.

The use of the non-member version allows one to write very generic methods.

Proposal for Guidelines
The use of non-member **`begin()`** and **`end()`** is encouraged.

static_assert and type traits

static_assert performs an assertion check at *compile*-time. If the assertion is true, nothing happens. If the assertion is false, the compiler displays the specified error message.

Example:

```
template <typename T, size_t Size>
class MyVector
{
   static_assert(Size > 3, "Size is too small");
   T points[Size];
};

int main()
{
   MyVector <int, 16> a1;
   MyVector <double, 2> a2;  // will produce compile error
   return 0;
}
```

Will produce (in my test program on the Mac):

```
sassert.cpp:12:5: error: static_assert failed "Size is too small."
    static_assert(Size > 3, "Size is too small.");
```

Note that since **static_assert** is evaluated at compile time, it cannot be used to check assumptions that depends on run-time values.

static_assert becomes more useful when used together with type traits. These are a series of classes that provide information about types at compile time. They are available in the **<type_traits>** header. There are several categories of classes in this header: helper classes, for creating compile-time constants, type traits classes, to get type information at compile time, and type transformation classes, for getting new types by applying transformation on existing types.

For example:

```
template <typename T1, typename T2>
auto add(T1 t1, T2 t2) G> decltype(t1 + t2)
{
   static_assert(std::is_integral<T1>::value,
                 "Type T1 must be integral");
   static_assert(std::is_integral<T2>::value,
                 "Type T2 must be integral");
   return t1 + t2;
}
```

Proposal for Guidelines

Type traits and static_assert is mostly for template class developer (class libraries). Since the use of templates in STAR is minimal these new C++11 features will be rarely used, if at all. There's no argument against using this feature if needed. Note that static_assert does not

violate STAR's messaging scheme since the assert error messages are printed at compile not run time.

constexpr - generalized and guaranteed constant expressions

One of the improvements in C++11, generalized constant expressions, allows programs to take advantage of compile-time computation. It is a feature that, if used correctly, can speed up programs.

The basic idea of constant expressions is to allow certain computations to take place at compile time—literally while your code compiles—rather than when the program itself is run.

Examples:

```
constexpr int multiply(int x, int y)
{
    return x*y;
}

constexpr int factorial(int n)
{
    return n <= 1 ? 1 : (n * factorial(n-1));
}
```

Then the compiler will evaluate the following statements at compile time instead at run time:

```
const int val multiply(10,10);
const int n5 factorial(5);
```

Another benefit of constexpr, beyond the performance of compile time computation, is that it allows functions to be used in all sorts of situations that previously would have called for *macros*. For example, let's say you want to have a function that computes the the size of an array based on some multiplier. If you had wanted to do this in C++ without a constexpr, you'd have needed to create a macro since you can't use the result of a function call to declare an array. But with constexpr, you can now use a call to a constexpr function inside an array declaration.

Example:

```
constexpr int defaultArraySize(int multiplier)
{
    return 10*multiplier;
}
```

and in the program it is now possible to use:

```
int array[defaultArraySize(3)];
```

Note that a **`constexpr`** specifier used in an object declaration implies const. A **`constexpr`** specifier used in an function declaration implies inline. If you declare a class member function to

be **constexpr**, that marks the function as **const** as well. If you declare a variable as **constexpr**, that in turn marks the variable as **const**. However, it doesn't work the other way--a **const** function is not a **constexpr**, nor is a **const** variable a **constexpr**.

There are also some limitations:
- It must consist of single return statement (with a few exceptions)
- It can call only other constexpr functions
- It can reference only constexpr global variables

You can make any object a **constexpr**. In this case the constructor must be declared a **constexpr** as well as the method to be used. This would go to far to discuss here.

Proposal for Guidelines
There's no argument against using this feature if needed. I can imagine that many methods used in unpacking data might benefit from this. It should be encouraged to be used.

Extended integer types

There are five standard signed integer types
signed char
short int
int
long int
long long int
and five unsigned integer types
unsigned char
unsigned short int
unsigned int
unsigned long int
unsigned long long int

The standard does not define the actual size of each but only guarantees that their sizes are:

signed char = unsigned char ≤ short int = unsigned short int ≤ int = unsigned int ≤ long int = unsigned long int ≤ long long int = unsigned long long int

While these types are sufficient for most tasks, there are times where the *precise* size has to be defined. Already before C++11 these types were implemented in most compilers but C++11 makes it official. Three types are defined in header **<cstdint>**: Exact-Width Types, Minimum-Width Types, Fastest Minimum-Width Types. Here is there definitions:

Exact-Width Types

One set identify types with precise sizes. The general form is intN_t for signed types and uintN_t for unsigned types, with N indicating the number of bits. Note, however, that not all systems can support all the types. For example, there could be a system for which the smallest usable memory size is 16 bits; such a system would not support the int8_t and uint8_t types.

Minimum-Width Types

The minimum-width types guarantee a type that is at least a certain number of bits in size. These types always exist. For example, a system that does not support 8-bit units could define int_least_8 as a 16-bit type.

Fastest Minimum-Width Types

For a particular system, some integer representations can be faster than others. For example, int_least16_t might be implemented as short, but the system might do arithmetic faster using type int. So **<cstdint>** also defines the fastest type for representing at least a certain number of bits. These types always exist. In some cases, there might be no clear-cut choice for fastest; in that case, the system simply specifies one of the choices.

int8_t **int16_t** **int32_t** **int64_t**	signed integer type with width of exactly 8, 16, 32 and 64 bits respectively with no padding bits and using 2's complement for negative values (provided only if the implementation directly supports the type)
int_fast8_t **int_fast16_t** **int_fast32_t** **int_fast64_t**	fastest signed integer type with width of at least 8, 16, 32 and 64 bits respectively
int_least8_t **int_least16_t** **int_least32_t** **int_least64_t**	smallest signed integer type with width of at least 8, 16, 32 and 64 bits respectively
intmax_t	maximum width integer type
intptr_t	integer type capable of holding a pointer
uint8_t **uint16_t** **uint32_t** **uint64_t**	unsigned integer type with width of exactly 8, 16, 32 and 64 bits respectively (provided only if the implementation directly supports the type)
uint_fast8_t **uint_fast16_t** **uint_fast32_t** **uint_fast64_t**	fastest unsigned integer type with width of
uint_least8_t **uint_least16_t** **uint_least32_t** **uint_least64_t**	smallest unsigned integer type with width of at least 8, 16, 32 and 64 bits respectively
uintmax_t	maximum width unsigned integer type
uintptr_t	unsigned integer type capable of holding a pointer

Proposal for Guidelines
In general the standard integer types should be used since they are typically also the “fastest” types on the respective architecture. In cases where the exact size matters, e.g. unpacking of raw data the extended types the extended integer types can be used. The use of ROOT types, such as **Long64_t** is strongly discouraged unless it is absolute necessary (see also discussion below).

Notes on the use of ROOT Types

Proposal for Guidelines
ROOT defines a large set of portable (fixed size on any architecture) and unportable types. The extensive definitions available now in C++11 (see above) makes ROOT types redundant. In all cases the built-in types or the extended version should be used. Use of builtin-types makes the code more portable and in several cases faster. The only exceptions are data member in "persistent" classes (e.g. StEvent) under schema evolution. Otherwise use **int** instead **Int_t**, **float** instead of **Float_t**, **double** instead of **Double_t**, etc. Note that underscores also make code less readable.

Notes on the use of ROOT mathematical functions

Proposal for Guidelines
ROOT provides a rich set of mathematical functions often adapted from the old **`cernlib`** or, more recently, wrapped GSL functions. They are heavily used in STAR code. However, ROOT also provides a set of mathematical functions that are already defined in **`<cmath>`**. Their use is discouraged. Use **`sqrt()`** instead of **`TMath::Sqrt()`**, use **`log()`** instead of **`TMath::Log()`**, use **`sin()`** instead of **`TMath::Sin()`**, etc. In all cases ROOT uses anyway the built in functions,e.g. **`TMath::Sqrt()`** calls **`sqrt()`**; there is no rational reason to use **`TMath`** functions when the same functionality is available in the standard and defined in **`<cmath>`**.

Suffix return type syntax (extended function declaration syntax)

C++11's new return value syntax presents a another use for **auto**. In all prior versions of C and C++, the return value of a function absolutely had to go before the function:

```
int multiply (int x, int y);
```

In C++11, you can now put the return value at the end of the function declaration, substituting **auto** for the name of the return type.

```
auto multiply (int x, int y) -> int;
```

In the above example the use of the new syntax does not provide any advantage, in fact it makes it less readable. However, there are several cases were the new syntax is in fact the only way to make things work.

Consider:

```
template<class T, class U>
??? add(T x, U y)
{
      return x+y;
}
```

What can we write as the return type? It's the type of "x+y", of course, but how can we say that? First idea, use decltype:

```
template<class T, class U>
decltype(x+y) add(T x, U y) // scope problem!
{
      return x+y;
}
```

That won't work because x and y are not in scope.
The solution is put the return type where it belongs, after the arguments:

```
template<class T, class U>
auto add(T x, U y) -> decltype(x+y)
{
      return x+y;
}
```

We use the notation auto to mean "return type to be deduced or specified later."
The suffix syntax is not primarily about templates and type deduction, it is really about scope.

Proposal for Guidelines
The use of the new return type syntax is very useful in templates and in methods where the return type is the class itself. See also **`decltype`**. The new return syntax, however, is not as easy to read as the standard method and should only be used where necessary and to simplify the code. It should not be regarded as an alternative way of defining a simple function.
Use the suffix syntax only if absolutely required.

Delegating constructors

Many classes have multiple constructors, especially in STAR. Often, an empty constructor sets variables to either 0 or any default parameters, while the non-empty constructors are used to set the data member to the values provided. Other examples are constructors that take different arguments, depending on the context the class is used. Often only parts of all data that defined in the class are known at construction time.

In many cases this requires to either write constructors with semi-identical code making code maintenance difficult, or providing a private “init()” function that is called internally by the various constructors.

Here’s an example how it was done in C++98:

```
class A {
public:
   A(): num1(0), num2(0) {average=(num1+num2)/2.;}
   A(int i): num1(i), num2(0) {average=(num1+num2)/2.;}
   A(int i, int j): num1(i), num2(j) {average=(num1+num2)/2.;}
private:
   int num1;
   int num2;
   double average;
};
```

To at least keep the repetitions at a minimum often this is done:

```
class A {
public:
   A(): num1(0), num2(0) {init();}
   A(int i): num1(i), num2(0) {init();}
   A(int i, int j): num1(i), num2(j) {init();}
private:
   int num1;
   int num2;
   double average;
   void init(){ average=(num1+num2)/2.;};
};
```

This revision eliminates code duplication but it brings the following new problems:

- Other member functions might accidentally call init(), which causes unexpected results.
- After we enter a class member function, all the class members have already been constructed. It's too late to call member functions to do the construction work of class members. In other words, **`init()`** merely reassigns new values to C’s data members. It doesn’t really initialize them.

Verbosity hinders readability and repetition is error-prone. Both get in the way of maintainability. So, in C++11, we can define one constructor in terms of another:

```
class A {
public:
   A(): A(0){}
   A(int i): A(i, 0){}
   A(int i, int j){
      num1=i;
      num2=j;
      average=(num1+num2)/2.;
   }
private:
   int num1;
   int num2;
   double average;
};
```

Delegating constructors make the program clear and simple. Delegating and target constructors do not need special labels or disposals to be delegating or target constructors. They have the same interfaces as other constructors. A delegating constructor can be the target constructor of another delegating constructor, forming a delegating chain. Target constructors are chosen by overload resolution or template argument deduction. In the delegating process, delegating constructors get control back and do individual operations after their target constructors exit.

Cons:

If not careful one can generate a constructor that delegates to itself:

```
class C
{
  public:

  C(int) { }
  C() : C(42) { }
  C(char) : C(42.0) { }
  C(double) : C('a') { }
};

int main()
{
  C c('b');
  return 0;
}
```

Here clang submits an error, gcc compiles and crashes with stack overflow.

Remark:

The example we used two above could be also solved using default arguments,

```
class A {
public:
     A(int i=0, int j=0){
      num1=i;
```

```
      num2=j;
      average=(num1+num2)/2.;
   }
private:
   int num1;
   int num2;
   double average;
};
```

which for this example might be even more elegant. This does not always work but it should be noted. C++11 has many ways to skin a cat.

Note that above code is attached to the class definition only for demonstration purposes. In STAR the use of code in class declarations is strongly discouraged since it reduces readability.

Proposal for Guidelines

The use of delegating constructors is ok. Be aware of self delegation.

Attributes

``Attributes'' is a new standard syntax aimed at providing some order in the mess of facilities for adding optional and/or vendor specific information (GNU, IBM, …) into source code (e.g. **`__attribute__`**, **`__declspec`**, and **`#pragma`**). As such their use for the common (STAR) programmer is limited. An attribute can be used almost everywhere in the C++ program, and can be applied to almost everything: to types, to variables, to functions, to names, to code blocks, to entire translation units, although each particular attribute is only valid where it is permitted by the implementation.

`[[noreturn]]`	Indicates that the function does not return. This attribute applies to function declarations only. The behavior is undefined if the function with this attribute actually returns.
`[[carries_dependency]]`	Indicates that dependency chain in release-consume std::memory_order propagates in and out of the function, which allows the compiler to skip unnecessary memory fence instructions. This attribute may appear in two situations: 1) it may apply to the parameter declarations of a function or lambda-expressions, in which case it indicates that initialization of the parameter carries dependency into lvalue-to-rvalue conversion of that object. 2) It may apply to the function declaration as a whole, in which case it indicates that the return value carries dependency to the evaluation of the function call expression. This attribute must appear on the first declaration of a function or one of its parameters in any translation unit. If it is not used on the first declaration of a function or one of its parameters in another translation unit, the program is ill-formed; no diagnostic required.
`[[deprecated]](C++14)` **`[[deprecated("reason")]](C++14)`**	Indicates that the use of the name or entity declared with this attribute is allowed, but discouraged for some reason. This attribute is allowed in declarations of classes, typedef-names, variables, non-static data members, functions, enumerations, and template specializations. A name declared non-deprecated may be redeclared deprecated. A name declared deprecated cannot be un-deprecated by redeclaring it without this attribute.

For example:

```
void f [[ noreturn ]] ()    // f() will never return
{
      throw "error";   // OK
}

int* f [[carries_dependency]] (int i); // hint to optimizer
int* g(int* x, int* y [[carries_dependency]]);
```

An attribute is placed within double square brackets: **`[[ ... ]]`**. **`noreturn`** and **`carries_dependency`** are the two attributes defined in the C++11 standard with **`deprecated`** being scheduled for C++14. Other attributes are in discussion to support MP.

Proposal for Guidelines
The use of attributes is discouraged in STAR.
There is a reasonable fear that attributes will be misused. The recommendation is to use attributes to *only* control things that do not affect the meaning of a program but might help detect errors (e.g. [[noreturn]]) or help optimizers (e.g. [[carries_dependency]]).