Discussion of New C++11 Features

Internal Note for STAR Coding Guideline Committee

Monday, November 24, 2014

Thomas Ullrich, Anselm Vossen, Mustafa Mustafa

ABSTRACT

This is a compilation of C++ features added in the new C++11 standard. This is a working document which discussed and explains most of them. Much of this was used to compile the new STAR Coding Guidelines but this document contains more details. Might be useful for later. Keep for records.

1. 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;
```

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.

3. 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(is_integral<T1>::value,
                 "Type T1 must be integral");
   static_assert(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.

4. 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.

5. 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).

6. 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.

7. 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>`**.

8. 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.

9. 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.

10. 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]]).

11. null pointer: nullptr

C++ 11 introduces the literal **`nullptr`**. It is of type **`nullptr_t`** and it converts implicitly to all pointer types. **`nullptr`** is a prvalue (i.e. pure revalue, you e.g. cannot take its address)

Example:

```
int* x=nullptr;
myclass* obj=nullptr;
```

Before, the alternative was to use **`NULL`** or **`0`** which did not have a pointer type. This could introduce problems with overloaded functions or templates.

Example:

```
void func(int i);
void func(float* f);
func(NULL);              // calls func(int i)
```

So the advantage of using **`nullptr`** is that the rvalue in pointer expressions has the correct type and it is obvious in the code that your lvalue is a pointer.

Proposal for Guidelines:
Use nullptr instead of **`0`** or **`NULL`** when assigning a null pointer.

12. Scoped and strongly-typed enums

C++11 introduces strongly typed enums. They are what the name suggests. Enums with a type:

Example:

```
enum class strongEnum { one, two, three, four};
```

So the syntax stays similar and the class keyword is added. Additional, the underlying type can be specified (otherwise defaults to int or, if e.g. large values are defined, system default)

Example:

```
enum class strongEnum : int{first, second};
```

The advantage of strong enums is that they are strongly typed and have a scope. With weak enums the problem was e.g.

Example:

```
enum weakEnum_color{red, green,blue};

// Error: redefinition of enumerator red and green
enum weakEnum_trafficLight{red, green, yellow};

// but this works:

enum class strongEnum_color{red, green,blue};
enum class strong_trafficLight{red, green, yellow};
strong_trafficLight trafficLight=strong_trafficLight::red;

// This does not work, wrong type
strong_trafficLight trafficLight=strongEnum_color::red;

// Error, since it doesn't convert by default to int anymore
.
// This is true even if the underlying type is int.
int c=strongEnum_color::red;

// but you can do (e.g. necessary if you want to use cout)
int c=static_cast<int>(strongEnum_color::red);
```

Strong enums solve the long standing problem with enums and scope. The strong type is also an advantage. In addition strong enums allow forward declaration and a definite size.

Proposal for Guidelines:
Use strong enums.

13. Lambdas

C++11 added Lambda functions (closures). This concept usually known from functional programming allows the creation of anonymous inline functions. Lambda functions can access variables of the enclosing functions (variable capture) either by value or by reference.

Lambdas are especially valuable with the std libraries iteration functions:

Examples:

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);

// Iterate over vector and print out each element
for_each(begin(v),end(v),[](int n)
{cout<<n<<endl;});

// Declare is_odd to be lambda function that returns true
// if the argument is odd. Note the return syntax.
// The return type can be omitted. It is then
// determined as if auto is applied to the statement
// after return.

auto is_odd=[](int n)->bool{return n%2==true;};
auto  pos=find_if(begin(v),end(v), is_odd);
//  the below snipplet returns 1 (the first position with an
//  odd number
if(pos!=end(v)) cout<<*pos<<endl;
```

A lambda function is of type **`function`**, so instead of **`auto`**, the above return type for **`is_odd`** could be written as **`function<bool(int)> is_odd;`**

Variable Capture

A powerful feature of lambda functions is that variables of the surrounding function can be captured. This is what the [] part of the function is for.

- **`[]`** No capture
- **`[=]`** capture by copy
- **`[&]`** capture by reference
- **`[a,b]`** capture only **`a, b`** by copy
- **`[&a, b]`** capture **`a`** by reference, b by copy

If the function is also modifying the variables that are captured by copy or wants to access the non--const members of a captured object, the lambda function has to be declared as 'mutable'. Note that this does not modify the captured variable in the surrounding function

Example (Gives output "a is now 5"):

```
int a=5;
auto testCopy=[a]() mutable {a=7;};
testCopy();
cout <<"a is now " << a <<endl;
```

Example (Gives output "a is now 7"):

```
int a=5;
auto testCopy=[&a]() {a=7;};
testCopy();
cout <<"a is now " << a <<endl;
```

Example does not compile (mutable missing):

```
int a=5;
auto testCopy=[a](){a=7;};
testCopy();
cout <<"a is now " << a <<endl;
```

Obviously, capturing variables by reference has side--effects.

Another danger in capturing variables by reference is that the reference might not be valid anymore when the lambda function is called:

Example:

```
function<int (float)> retLambda(int a, int& b)
{
      int c=5;
      // If this is & (i.e. pass by reference,
      // it doesn't work... (a,b, are nonsense)
      return [&] (float d) ->int {cout <<"lambda param: "<< d
      <<", a: "<< a <<" b : " << b << " c: "<< c << endl;
      b=7;
      cout <<" lambda b is now " << b <<  endl;
      return a+b+c;
}



void testVarByReference()
{
      int a=1; int b=2; int c=3.0;
      auto f=retLambda(a,b);
      cout <<" lambda returns " << f(c) << endl;
      cout <<" b is now " << b << endl;
}
```

In the above example there are several things that are noteworthy:

- The function that returns a lambda function has the return type **`function<int(float)>`**
- All variables are captured by reference. However when the lambda function **`f`** is called

in **`testVarByReference`**, all variables that where on the stack of the function that generated the lambda function (**`retLambda`**) are invalid. That means that **`c`** and **`a`** are invalid.

- The variable that was given by reference, **`b`**, is still valid. In fact it is modified in the lambda function, so that it has the value 7 after the lambda function is called.

Proposal for Guidelines:
Lambda functions are a powerful addition. They can be very useful in particular when iterating over containers. They should not be used for regular functions as there might be a performance penalty when they are called often.
Capture by reference can lead to side--effects. Discourage capture by reference.

14. Initializer lists (uniform and general initialization)

C++ 11 introduced general initializer lists using initializer_lists. One can then use the **`{}`** syntax with containers to pass an initializer list. This solves several problems

- Initializations are now uniform in syntax
- Containers can be initialized easily
- Dynamically allocated arrays of plain old datatypes can be initialized easily
- Class members can be initialized

Example for class member initializations and initializations of various types:

```
class C1
{
public:
	//x is initialized with the list of values
	C1(): x{0,1,2,3} {}
private:
	int x[4];
};

class C2
{
public:
	C2();
private:
	int x=7; //class member initializer  (not possible in C++03)
};


class C3
{
public:
	C3();
private:
    //class member initialization with list
    int y[5] {1,2,3,4};
};

class C4
{
public:
    C4();
private:
    // () doesn't work...
    // class member initializations allow ‘old’ syntax’
    string s{"abc"};
    string s1=(“abc");
    double d=0;
    char * p {nullptr};
    int y[5] {1,2,3,4};
};

class C5
{
public:
    C5(){} // x initialized to 7 when the default ctor is invoked
    C5(int y) : x(y) {} //overrides the class member initializer
private:
    int x=7; //class member initializer
};

C5 c; //c.x = 7
C5 c2(5); //c.x = 5

//C++11: default initialization using {}
int n{}; //zero initialization: n is initialized to 0
int *p{}; //initialized to nullptr
double d{}; //initialized to 0.0
char s[12]{}; //all 12 chars are initialized to '\0' string s3{};
//same as: string s3;
char *p2=new char [5]{}; // all five chars are initialized to '\0'

//sequence constructor vector<int> vi {1,2,3,4,5,6};
vector<double> vd {0.5, 1.33, 2.66};

int a{0};
string s4{"hello"};
string s5{s4}; //copy construction
vector <string> vs{"alpha", "beta", "gamma"};
map<string, string> stars
    { {"Superman", "+1 (212) 545-7890"},
    {"Batman", "+1 (212) 545-0987"}};
double *pd= new double [3] {0.5, 1.2, 12.99};
```

but:
```
      double *pd= new double [3] {0.5, 1.2, 12.99, 4};
```
leads to run--time (not compile time) exception. Initializer list doesn't match allocated space.

This works:
```
      double pdp[] {0.5, 1.2, 12.99,30};
```

Proposal for Guidelines:
Use new initializer lists and ideally do this consistently. Only pitfall I see is the use with dynamically allocated arrays. Here it would be interesting to see if the new static code checker catches these instances.

15. Alignments

C++ introduces the **`alignof`** operator and **`alignas`** specifier. Before (C++03) it was not possible to align storage explicitly. However, most compilers provided directives that made this possible. These features are mainly needed for performance issues. As such it is probably not relevant for "everyday" users code.

The alignof operator returns the alignment in bytes that is required of an instance of the given type. The alignas specifier, specifies the alignment of the given variable either in byte or as a given type.

Example:

```
struct Empty {};

struct Foo {
      int f2;
      float f1;
      char c;
};

cout << "alignment of empty class: " << alignof(Empty) << '\n'
     << "alignment of pointer : " << alignof(int*) << '\n'
     << "alignment of char : "   << alignof(char)<< '\n'
     << "alignment of Foo : " << alignof(Foo) << endl;

alignment of empty class: 1
alignment of pointer : 8
alignment of char : 1
alignment of Foo : 4

alignas(double) unsigned char c[1024]; // array of characters,
                                            // suitably aligned for doubles
alignas(16) char c2[100]; // align on 16 byte boundary
```

There is also the align function, that can be used to return a pointer to an address within a given piece of allocated memory that is properly aligned

Example:

```
template <std::size_t N>
struct MyAllocator
{
      char data[N];
      void* p;
      std::size_t sz;
      MyAllocator() : p(data), sz(N) {}
      template <typename T>
      T* aligned_alloc(std::size_t a = alignof(T)) {
            if (std::align(a, sizeof(T), p, sz)) {
                  T* result = reinterpret_cast<T*>(p);
                  p = (char*)p + sizeof(T);
                  sz -= sizeof(T);
                  return result;
            }
            return nullptr;
      }
};

MyAllocator<64> a;

// allocate a char
char* p1 = a.aligned_alloc<char>();
if (p1) *p1 = 'a';
cout << "allocated a char at " << (void*)p1 << '\n';

// allocate an int
int* p2 = a.aligned_alloc<int>();
if (p2) *p2 = 1;
cout << "allocated an int at " << (void*)p2 << ‘\n';
// allocate an int, aligned at 32-byte boundary
int* p3 = a.aligned_alloc<int>(32);
if (p3) *p3 = 2;

cout << "allocated an int at "
     << (void*)p3 << " (32 byte alignment)\n";
```

There is also the aligned union (existed before C++ 11).
It can be used to define a union which is aligned according to the stricter alignment requirements of the given types. I.e. it is aligned for the all types that make up the union.

Example:

```
struct S1
{
      std::string file;
};

struct S2
{
      S2(const string &from, const string &to)
            : from {from}, to{to} {}
      string from;
      string to;
};
// Declare a variable of the type aligned_union that
// has at least sizeof(S2) space and is aligned for S1
// and S2 (uses template parameter packs (varidadic templates))

aligned_union<sizeof(S2), S1, S2>::type storage;

S2 * p = new (static_cast<void*>(addressof(storage))) S2("from",
"to");

// ...

p->~S2();
```

Proposal for Guidelines:
For portability it the `alignas`, `alignof` and align features should be preferred over compiler directives. However, these features are only needed in very specific circumstances. I would therefore propose not to mention them in the guidelines.

16. Inline namespace

C++ 11 introduces inline namespaces. An inline namespace is basically a namespace that has an implicit using directive before its declaration. Its main (and only?) use is for versioning.

Example:

```
namespace inner_ordinaryNS
{
      int ordinary=3;
}

inline namespace inner_inlineNS
{
      int nonOrdinary=4;
}
```

```
// If inline is missing compiler complains
//(cannot be reopened as non-inline NS)
inline namespace inner_inlineNS
{
      int ordinary=6;
}

void testInlineNS()
{
      cout <<"ordinary from inline is: "<< ordinary << endl;
      //hides inline namespace definition int ordinary=2;
      //can still access it with explicit scope
     cout <<"now ordinary is : " << ordinary << "inline ordinary:"
          << inner_inlineNS::ordinary << endl;
     int nonOrdinary=5;
}
```

Apparently the use case is something like this

```
#ifdef  __VERSION_1
      inline
#endif
namespace  version1_NS
{
}
```

to use in versioning. A similar behavior could be achieved by doing:

```
#ifdef  __VERSION_1
      using  version1_NS
#endif
```

However, there are some loopholes that would break this code. E.g. if one wants to specialize templates one has to know the namespace.

Proposal for Guidelines:
No reason not to use inline namespaces for versioning. However, their use is limited enough that we might not want to mention it

17. decltype

`decltype` can be used to get the type of a variable. Its simple use is very similar to the `auto` statement

Example:

```
int  x=7;
decltype(x)  y=x;
```

```
//same thing auto y2=x;
```

It can be used for lambda functions, but here auto works as well:

Example:

```
auto f = [] (int a, int b) -> int { return a*b;};

decltype(f) f2=f;   //works as well, so why use decltype here?

auto f3=f;
```

The one use case I found is to declare a return type that is part of a template parameter. Example:

```
class A
{
public:
      int makeObject() const {return 8;};
      A(){};
};

template <typename Builder>
auto
makeAndProcessObject (const Builder& builder) ->
decltype(builder.makeObject())
{
      auto val = builder.makeObject();
      // do stuff with val return val;
}
```

Proposal for Guidelines:
Do not use `decltype` if you can use `auto`. `decltype` is useful for templates, since there is no other way to declare the return type of a function if the return type depends on the parameter. I suspect that the use cases are limited at STAR though.

18. Range-for statement

C++ syntax is extended to support easier iteration over a list of elements. For example:

Try it live.

```
int main() {
    vector<int> v {1,2,3,4,5};
    for(auto& i:v)  // reference to element
    {
         cout<<i<<endl;
         i +=1;        // modifies element value
    }

    for(auto i:v)   // copy of element
    {
         cout<<i<<endl;
    }
    return 0;
}
```

Range-for statements work for any type where `begin()` and `end()` are defined and return iterators.

The use of range-for loops increases code readability. However, programmers often need both, the elements of an iterable collection and its index. This is not directly supported in C++11.

Unless optimized away by compiler, using a copy of element could come at a performance cost if the element type is large.

Proposal for Guidelines:
- Range-for loops are useful and should be allowed.
- Ordinary loops should be preferred when programmer needs the element index. Avoid having your own counter.
- Using reference to elements is encouraged when dealing with large objects. Use `const` reference when you don't need to modify the object state.

19. Override controls: override

No explicit keyword is needed to to override a virtual function in C++98. In C++11, keyword `override` has been added to make the programmer's intention explicit to the compiler and reader. For example:

Try it live.

```
struct B  {
      virtual void a(int);
      virtual void f();
      virtual void g() const;
      void k(); // not virtual
      virtual void h(char);
};

struct D : B {
      void a(float) override; // doesn't override B::a(int)
                              // (wrong signature)
      void f() override; // overrides B::f()
      void g() override; // doesn't override B::g() (wrong type)
      void k() override; // doesn't override B::k()
                         // (B::k() is not virtual)
      void h(char); // overrides B::h()
};
```

Error given by gcc compiler when there is a problem with an override attempt is

```
error: 'void foo::foo()' marked override, but does not override.
```

The case of the `D::h(char)` is curious. gcc doesn't give a warning when the `override` keyword is not used. This is likely to avoid a volcano of warnings when compiling older code.

The `override` keyword is very useful. It makes the code more readable by making the programmer intention explicit and avoids potential problems by catching, otherwise silent errors, at compile time. Common bugs such as missing a function constant'ness or being careless with function signature when overriding can be avoided.

Proposal for Guidelines:
- Use of `override` keyword is encouraged.

20. Override controls: final

`final` keyword can be used to prevent inheriting from classes or simply preventing overriding methods in derived classes. For example:

Try it live.

```
struct Base1 final { };
struct Derived1 : Base1 { }; // ill-formed because the class Base1
                             // has been marked final
struct Base2  {
    virtual void f() final;
};
struct Derived2 : Base2  {
    void f(); // ill-formed because the virtual function
              // Base2::f has been marked final
};
```

Using `final` closes the possibility of better implementation of functions in derived classes.

Proposal for Guidelines:
No recommendation.

21. Control of defaults: default and delete

In C++11, the programmer can instruct the compiler not to create certain defaults by using the specifier = `delete`. This is particularly useful in two cases:

1. Making objects non-copyable:

```
struct NonCopyable  {
     NonCopyable() = default;
     NonCopyable(const NonCopyable&) = delete;
     NonCopyable & operator=(const NonCopyable&) = delete;
};
```

2. Preventing implicit conversion of function arguments:

```
struct NoInt  {
     void f(double i);
     void f(int) = delete;
};
```

The specifier = `default` can be used to state the programmers intention:

```
struct SomeType  {
    SomeType() = default; // The default constructor
                          // is explicitly stated.
    SomeType(OtherType value);
};
```

However, the verbosity here is redundant, it is useful as a declaration of intention.

For classes, the default generated functions are always public. Programmer can control the visibility of the defaults by using = `default`.

The current STAR guidelines state that "each class should have an assignment operator and a copy constructor" which forces programmers to implement their own copy functions even when they want the default behavior. The specifier `default` casts this guideline and avoids the pitfalls of implementing copy/assignment constructors with default behavior.

Proposal for Guidelines:
The current STAR guideline should be changed to:
- Each class should have an assignment operator and a copy constructor or defaults should be explicitly requested.
- Do not implement your own copy/assignment when member-wise copy is desired.
- *Implement your own copy/assignment when class is a resource-handler or delete them.*
- No guidance on other uses of = `delete`.

22. Control of defaults: move and copy:

In addition to constructors, control of defaults can be used for copy/move assignment operators or constructors and destructor. However, one should pay attention to the Rule of Five. Stated roughly by Stroustrup:

1. If any move, copy, or destructor is explicitly specified (declared, defined, =default, or =delete) by the user, no move is generated by default.
2. If any move, copy, or destructor is explicitly specified (declared, defined, =default, or =delete) by the user, any undeclared copy operations are generated by default, but this is deprecated, so don't rely on that.

This means that for backward compatibility STAR rule of mandating definition of copy assignment operator and constructor means that move semantics will *not* be generated by default. This is good.

Now, if we maintain the STAR guideline on copy assignment/constructor it will be a good practice to ask programmers to be explicit about their desires for the move counterparts (that is if we eventually allow the move semantics).

Proposal for Guidelines:
- Pay attention to the Rule of Five.
- Be explicit about your intentions for the move semantics.

23. In-class member initializers

C++98 allows in-class member initialization for static members only. C++11 allows in-class member initialization for any variable. For example:

```
Class foo {
```

```
public:
    int x = 1;
}
```

This is basically equivalent to using initialization lists in constructors. The advantage of in-class initialization is that it allows consistent default initialization when there are multiple constructors and saves a lot of typing resulting in cleaner codes.

Constructor initialization overrides in-class initialization.

Proposal for Guidelines:
- In-class member initialization should be encouraged.

24. RVO and copy elision

This is a note that will become useful later.
Return value optimization (RVO) and copy elision are both implemented in gcc compiler.
The copy elision as an implementation of the *as-if* rule of the standard which states that a compiler is allowed to do optimization to avoid unnecessary copying if the behavior of the program is the same *as if* all the requirements of the standard has been fulfilled. Furthermore,

The term return value optimization refers to a special clause in the C++ standard which goes even further than the "as-if" rule: an implementation may omit a copy operation resulting from a return statement, even if the copy constructor has side effects.

Example:

Try it live.

```
#include <iostream>
struct C  {
    C() {};
    C(const C&) { std::cout << "A copy was made.” << endl; }
};

C f() {   return C(); }

int main() {
    std::cout << "Hello World!\n";
    C obj = f();
}
```

Compiling and running with gcc gives:

```
Hello, World!
```

So both copies (from local variable to the stack and from the stack to obj) have been optimized away. i.e. RVO is being used even though avoiding the copy constructor has side effect.

You can avoid this optimization by using the gcc flag `-fno-elide-constructors`.

25. Rvalue reference and move semantics

One can find multiple practical definitions of lvalueness or rvalueness. For our purposes, the following is sufficient:

- If you can take its address using the built-in address-of operator (&) then it is an lvalue, otherwise, it is an rvalue. OkTo a lesser degree the if-it-has-a-name rule is the if-it-has-a-name rule is Although it is not completely true, the if-it-has-a-name rule is very useful:
- If it has a name then it is an lvalue, otherwise, it is an rvalue.

Rvalue reference is designated with an && as opposed to & for lvalue reference. Here is an example of function overloading to handle lvalue and rvalue arguments separately:

Try it live.

```
#include <iostream>
using namespace std;

int foo() {
    return 5;
}

void print(int const& x) {
    cout << __PRETTY_FUNCTION__ << endl;
    cout << x<<endl;
}

void print(int&& x) {
    cout << __PRETTY_FUNCTION__ << endl;
    cout << x << endl;
}

int main() {
    std::cout << "Hello, World!\n";
    int x =6;
    print(x);  // call print on an lvalue
    print(foo()); // call print on an rvalue
}
```

Now this looks cool. However, the real power of the ability to distinguish between rvalues and lvalues in C++11 is to enable two things: 1) move semantics 2) perfect forwarding.

Move semantics:
The move semantics allow to get rid of expensive copies from temporary (rvalue) objects when a move is intended. Now that we can detect temporary objects using rvalue references we can overload the copy/assignment functions to do the less expensive move from the temporary object by simply pointing the current object's pointers to the temporary object's resources and nullifying the latter's pointers. To add to the multitude of examples of move semantics implementation here is one:

Try it live.

```
#include <cstddef>
#include <iostream>

using namespace std;

class dataHandler {
private:
    int* mData;

public:
    dataHandler(int x=-999): mData(new int(x)) {
         cout << __PRETTY_FUNCTION__ << endl;  // copy constructor
    }

    dataHandler(const dataHandler& x): mData(new int(x.data())) {
         cout << __PRETTY_FUNCTION__ << endl;  // copy assignment 
operator
    }

    dataHandler& operator=(const dataHandler& rhs) {
         cout << __PRETTY_FUNCTION__ << endl;
         if(this == &rhs) return *this;
         *(this->mData) = *(rhs.mData); // copy value
         return *this;     // move constructor
    }

    dataHandler(dataHandler&& x) {
        cout << __PRETTY_FUNCTION__ << endl;
        this->mData = x.mData;
        x.mData = nullptr;         // move assignment operator
    }

    dataHandler& operator=(dataHandler&& rhs)  {
        cout << __PRETTY_FUNCTION__ << endl;
        if(this == &rhs) return *this;
        this->mData = rhs.mData;
        rhs.mData = nullptr;
        return *this;
    }

    ~dataHandler() {
        cout <<__PRETTY_FUNCTION__ << endl;
        if(mData) delete mData;
    }

    inline int data() const { return *mData;}
    inline void data(int const d) { *mData = d;}
};

dataHandler get_a_dataHandler() {
    dataHandler t(6);
    return t;
}

int main() {
    cout << "Testing move semantics…" << endl;
    dataHandler t0;
    t0 = get_a_dataHandler(); // should call move assignment operator.
    dataHandler t1(get_a_dataHandler()); // should call move
                                         // constructor (unless
                                         // optimized away.
                                         // See RVO/elision).
    cout << "... GoodBye! …" << endl;
    return 0;
}
```

An rvalue reference itself is not necessarily an rvalue. For example, inside the move constructor the variable `x` is an rvalue reference, but it is an lvalue (you can take its address, it has a name). This case is important when one wants to construct base classes in a move function of the derived class. The base class move function should be invoked and this can be achieved by statically casting the variable x to an rvalue reference, i.e. hiding its name which can be acheived using std::move.

```
Derived(Derived&& rhs)
   : Base(rhs) // wrong: rhs is an lvalue
{
    // Derived-specific stuff
}

Derived(Derived&& rhs)
   : Base(std::move(rhs)) // good, calls Base(Base&& rhs)
{
    // Derived-specific stuff
}
```

std::move hides the name of its arguments (static casting to an rvalue reference).

Perfect forwarding:
There is one subtlety with rvalues and deduced types. The rvalueness/lvalueness of a deduced type follows that of the initializer. For example, in a function template:

```
template<typename T>
void print(T&& x) { cout << x << endl;}
```

So calling print on a lvalue makes `x` an lvalue reference, same if `x` is an rvalue. Now when does this matter? It doesn't matter inside `print` itself since `x` is an lvalue there anyway. It matters when you want to pass `x` to another function, do you pass it as an lvalue (just `x`) or hide its name using std::move? The answer obviously depends on the nature of `x`, you want to preserve that. This can be achieved using std::forward. std:forward allows rvalue references as rvalues and lvalue references as lvalues.

Proposal for Guidelines:
- Classes should have move constructor and assignment operator OR explicitly delete them.
- Strive to define your move semantics so that they cannot throw exceptions and declare them so using `noexcept`.
- Use std::move to pass argument to base classes in move constructor and assignment operator.
- Use std:forward to forward arguments to classes constructors in templated functions or classes.
- Remember that an rvalue reference is not necessarily an rvalue itself.

- Take advantage of RVO/elision, don't be afraid to return by value.

26. Smart pointers

It is a modern C++ idiom to get rid of naked pointers whenever possible. However, it is currently difficult to devise an error free scheme where smart pointers can live in harmony with ROOT object ownership and management rules. For example:

```
#include <iostream>
#include <memory>
#include "TH1F.h"
#include "TFile.h"

int main() {
    TFile f("out.root", "recreate");
    f.cd();
    std::unique_ptr<TH1F> h {new TH1F("h", "h", 100, -5, 5)};
    h->FillRandom("gaus", 10000);
    h->Write();
    f.Close();
    return 0;
}
```

The histogram which is handled by a unique pointer was owned by the current gDirectory. Since I politely closed the file before I exit my program the histogram was destroyed by ROOT memory management guy. Now at the end of main() my pointer goes out of scope and its resource needs to be freed, but it has already been freed!

I also imagine that STAR "WhiteBoard" of StMakers won't be easy to marry with smart pointers either.

Proposal for Guidelines:
- Do not use smart pointers.

References

Most of the examples in this document are modified versions of those in:
- C++11 wiki page.
- Stroustrup's C++11 FAQ.