C++11 usage in Pythia 8.3

Objectives

Objectives for this discussion session:

1. Go through new features of C++11 and discuss them to make sure we are more or less on the same page.
2. Decide if we want to endorse each feature in the coding style guidelines.
3. Decide how to effect these decisions.

Levels of endorsement

- Encourage means use this feature if at all possible.
  Example: use `int` for all integers, even if `char` saves space
- Allow means that the feature may be used, and it depends on the judgment/preference of the developer.
  Example: whether to pass by pointer or by reference
- Discourage means the feature should generally be avoided, but can be allowed in situations where it would be very helpful.
  Example: using `char*` for strings.
- Disallow means we don't allow developers to use a feature at all. In many cases, we disallow features because they would only add a small improvement over alternatives, but are conceptually difficult for people who are not familiar with them. Example: `typedef`

Minor features

- Encourage: Use `nullptr` for null pointers, not `NULL` or `0`.
- Encourage: Use `override` whenever applicable.
- Allow: For values that are truly constant, use `constexpr` instead of alternatives (such as `static const` or `#define`). Often this has little effect in practice, so it's a less strict requirement.
- Allow: Range-based for loops are generally good, but it can sometimes be a matter of taste.
- Disallow: We haven't used `final` anywhere, so it's inconsistent with style. It does not bring a lot of value unless it is used consistently.
- Disallow: `static_assert` is inconsistent with the Pythia style and the way we handle testing and errors. Hence, we don't believe it makes the code easier to maintain.

auto

The auto keyword is definitely useful. The following examples illustrates cases in which we Encourage, Allow, Discourage, or Disallow its usage.

1. `map<int,ParticleDataEntry>::iterator iter = pdt.find(id)`
2. `function<double(double)> integrand = [=](double x) { return f(x, a, b, c); }`
3. `for (PhysicsBase* physicsPtr : physicsPtrs)`
4. `for (Particle& p : pythia.event)`
5. `ParticleDataEntry* pde = particleDataPtr->find(id)`
6. `pair<Vec4, Vec4> ps = Rndm::phaseSpace2(...)`
7. `Particle& h = event[i]`

Conclusion: Encourage we want to use auto for very long names (especially iterators), or intermediate length names when the implied type is completely obvious from context. When in doubt, avoid using it.

rvalue references

It can be beneficial to have rvalue references as function parameters. As an example, consider

```
Hist::operator=(const Hist& h);
Hist::operator=(Hist&& h);
```

Writing something like `Hist h = h1 + h2` will use the rvalue version, which can be faster since it can use move operations instead of copy operations.

Conclusion: Allow for move constructors and move assignment operators, which should be implemented when custom copy constructors/operators are implemented. This can have a positive effect on performance, and usage in this situation is prevalent in C++. Disallow otherwise, since their meaning in other contexts is much less transparent.

Lambda functions

Lambda functions have some applications and can sometimes make the code a lot cleaner. Example:

```
bool integrateGauss(double& resultOut,
  function<double(double)> f, double xLo, double xHi);

// Define lundFF as lambda function of only z,
\\ fixing a, b, c, mT2 as parameters.
auto lundFF = [=](double z) {
    return LundFFRaw(z, a, b, c, mT2); };
check = integrateGauss(denominator, lundFF, 0, 1);
if (!check || denominator <= 0.) return -1.;
```

The biggest issue is that they are conceptually difficult.

Conclusion: Discourage, allow only when the alternative is significantly messier, such as having a base class with an abstract `operator()`. Good comments are required; consider using the word "lambda" to indicate a relevant search term for maintainers to find more information.

Smart pointers I - `shared_ptr`

A `shared_ptr` should be used when many objects share a resource and it's not clear ahead of time which object will release the resource last. This is especially useful when taking objects created by the user (e.g. an object derived from `UserHooks`), because it guarantees both that the object persists at least as long as the Pythia object, and that Pythia will release the reference later.

Conclusion: Encourage when the user passes custom objects to Pythia. Discourage otherwise. We would like to move away from raw pointers, especially `new`/`delete`, but this will require some effort and a bit of learning curve.

Smart pointers II - unique_ptr

A unique_ptr is a pointer that cannot be shared, but will delete itself automatically when it goes out of scope. One example is

```
class SigmaTotal {
  virtual ~SigmaTotal() {
    if (sigTotElPtr) delete sigTotElPtr; }
  SigmaTotAux*  sigTotElPtr;
};
```

If sigTotElPtr was a unique_ptr, the custom destructor would not be necessary. The problem is that it can be conceptually difficult to understand for someone who's not familiar with this use case.

Conclusion: Disallow. Smart pointers are not yet an established part of the Pythia style, and unique pointers don't add enough value to justify the learning curve. This might be revised in the future.