C++11 — What can we already use from the latest standard?

Ramses van Zon

SciNet HPC Consortium, University of Toronto

March 20, 2013

What is C++11?

- There is a C++ standards committee.
- The first C++ standard was accepted in 1998: C++98.
- An updated standard was accepted in 2003: C++03.
- Proposals for language extensions in 2005: TR1.
- New standard was finished in 2011: C++11

Some C++11 features already existed as compiler-specific extentions, but no compiler is fully C++11 compliant yet.

Aim of this talk:
Show some of the most commonly implemented C++11 features.
(will assume reasonable C++03 knowledge)

Tested C++11 compilers

- GNU's g++ version 4.7.2 (September, 2012)
- Intel's icpc version 13.1 (February, 2013)

(as these are available on SciNet's GPC and ARC)

List of supported C++11 language features can be found at

`http://wiki.apache.org/stdcxx/C++0xCompilerSupport`

Haven't found such a nice list for C++ library enhancements.

How to compile

g++

```
$ g++ -std=c++11 -pthread
```

icpc

```
$ icpc -std=c++11 -pthread
```

Not tested here, but for your information, the IBM compilers have a c++11 flags too:

```
$ xlc++ -qlanglvl=extended0x
```

xlc++ not tested much because in aix it lacks many new libraries, and in linux is harder to make work with newer gcc's.

Language enhancements

- With the C++11 standard, core C++ has changed substantially
- Different pieces of the language fit together better
- Many new features.

We will look at the language extensions first, and consider the library extensions after.

Language enhancements

In particular let's consider:

- Helpful language extensions
- Static asserts
- Range-based for
- Initialization
- Move semantics
- Lambda expressions

Helpful language extensions

auto Placeholder for a type that can be deduced:

```
auto i = 4; //same as "int i = 4"
auto d = 0.4; //same as "double d = 0.4"
auto p = std::make_pair(i,d);
//std::pair<int,double> p = std::make_pair(i,d);
```

decltype Type of a given expression:

```
decltype(std::make_pair(i,d)) p;
p = std::make_pair(i,d);
```

Useful with new function declaration syntax for deduced return types:

```
template <class A, class B>
auto add(const A&a, const B&b) -> decltype(a+b)
{
   return a+b;
}
```

Helpful language extensions

extern template Allows templates to be instantiated in only one translational unit, not in every unit that uses it.

```
template <typename T> class C {
   ...
};
template class C<int>; //explicit instantiation
```

elsewhere

```
template <typename T> class C {
   ...
};
extern template class C<int>;//no instantiation
```

long long Finally in the standard: at least 64bit.
standard still ambiguous about the #bits in `int` *s,*
but in `<cstdint>` *there now are* `int8_t int16_t, int32_t, int64_t, uint8_t, ....`

nullptr always a pointer, unlike `0`, which is an `int` first.

Helpful language extensions

Right Angle Brackets So we can just write

```
std::vector<std::pair<int,double>> p;
```

OOP stuff: Delegating Constructors, Defaulted And Deleted Functions, **`override`** and **`final`**

Static asserts

With **assert** we can test for conditions at runtime.

static_assert does the same but at compile time.

```
static_assert(4!=5, "four is not five");
```

If the condition is not fulfilled, the compiler will throw an error and print the message give to **static_assert**

Range-based for loops

For containers, the standard loop is using iterators:

```
std::vector v(10);
for (std::vector::iterator i = v.begin();
      i != v.end();
      i++)
      //something with *i
```

Now we can do:

```
std::vector v(10);
for (auto i:  v)
   // something with i
```

This passes a copy by default, so modifying `i` does not change `v`.
To be able to do that, get a reference:

```
std::vector v(10);
for (auto & i:  v)
   // something with i
```

- Not for dynamic arrays.
- Does not cooperate with openmp.

Initialization

- C++03 has various ways to initialize (arrays of) objects.
- Inherited from C:

```
int i = 7;
float f[3] = {0.1,0.2,0.3};
struct R {
   int i;
   std::string s;
}
R r = {5,"Bill"};
```

- Using constructor:

```
struct R {
   int i;
   std::string s;
   R(int ii,const std::string &ss):i(ii),s(ss){}
}
R r(5,"Bill");
```

- Explicit assignment, e.g. `std::vector` and `new` ed arrays.
- C++11 make this more uniform.

Initialization

- Uniform initialization using curly brackets {}
- Constructor or struct can both use

```
R r {5,"Bill"};
```

  Curly brackets call the appropriate constructor if it exists.
- Can use with new too:

```
R* r = new R{5,"Bill"};
```

- Can use for arrays too:

```
R* r = new R[3] {{1,"Bill"},{2,"John"},{3,"Jane"}};
```

- And for containers:

```
std::vector<R> r {{1,"Bill"},{2,"John"},{3,"Jane"}};
```

  (not supported by icpc)

SciNet compute • calcul CANADA

Initializer lists

Consider this last case again:

```
std::vector<R> r {{1,"Bill"},{2,"John"},{3,"Jane"}};
```

Which constructor would this call?

In fact, this construction uses a new type of list: initializer list:

- Classes need a constructor that expects an initializer list for this to work. All STL containers should have this.
- If they do, these constructors will be called preferably over others when using curly brackets.
- Need to use the old `()` constructors if that's not wanted, e.g.

  ```
  std::vector<int > r(9);
  ```

  gives a vector of 9 elements, wheras

  ```
  std::vector<int > r{9};
  ```

  gives a vector with 1 elements, whose value is 9.

Move semantics

- In C++03, assigning a temporary object to a named object:

```
class C;
C get_a_C() {
   return C(1,0);
}
C c;
c = get_a_C();
```

  means a copy of the temporary has to be made.
- But what we really want is for that temporary to become the named object.
- In other words, we want to move the temporary into the variable `c`.
- This would be different from copying, because any memory in the object to be moved would not need to be reallocated, moved and freed.

Move semantics

- In C++03, assigning a temporary object to a named object:

```
class C;
C get_a_C() {
   return C(1,0);
}
C c;
c = get_a_C();
```

  means a copy of the temporary has to be made.
- But what we really want is for that temporary to become the named object.
- In other words, we want to move the temporary into the variable `c`.
- This would be different from copying, because any memory in the object to be moved would not need to be reallocated, moved and freed.
- By the way....

Move semantics

- In C++03, assigning a temporary object to a named object:

```
class C;
C get_a_C() {
   return {1,0}
}
C c;
c = get_a_C();
```

  means a copy of the temporary has to be made.
- But what we really want is for that temporary to become the named object.
- In other words, we want to move the temporary into the variable `c`.
- This would be different from copying, because any memory in the object to be moved would not need to be reallocated, moved and freed.
- By the way....

Move semantics

- Logically can only move things that are temporary, or "rvalues".
- rvalues are expressions that can only occur at the right hand side of an assignment (barring technicalities).
- C++11 can handle references to rvalues. The reference to an rvalue of type `T` is denoted by `T&&`.
- By defining a constructor and an assignment operator that take an rvalue reference (in addition to the usual ones), a class can implement move semantics.
- These should put the internal state of rhs into the lhs and modify the rhs to become 'deletable' without side effects for the lhs (ie., set all pointers to **`nullptr`**).
- Most STL classes should now be doing this.

Example

```
struct X {
   int * x;
   X():  x{nullptr} {}
   ~X() { delete x; }
   X(int i):  x{new int {i}} {}
   X(const X&o):  x{new int {*o.x}} {}
   X(X&&o):  x{o.x} { o.x = nullptr; }
   X& operator= (const X&o) {
      if (this != &o)
         x = new int {*o.x};
      return *this;
   }
   X& operator= (X&&o) {
      if (this != &o) {
         x = new int {*o.x};
         o.x = nullptr;
      }
      return *this;
   }
};
```

Example

```
X get_an_X() {
   return {7};
}
int main() {
   X a(5);
   X b(std::move(a));
   X c;
   c = std::move(b);
   X d;
   d = get_an_x();
}
```

Lambda expressions

- Sometimes you suddenly need a part of your code to become function.
- Example:

```
auto a = new double [5] {2.0,1.1,1.2,1.3,1.4};
double sum {0.0};
for (double * iter=a; iter!=a+5; iter++)
   sum += *iter;
```

To:

```
auto a = new double [5] {2.0,1.1,1.2,1.3,1.4};
double sum {0.0};
std::for_each(a,a+5, ????);
```

(std::for_each is in `<algorithm>`)

- `????` should be a function to be called for each.
- Lambda's are functions defined on the spot.

Lambda expressions

- General strucure:

```
[capture](arguments) -> return-type
{
   body
}
```

- Like a new style function without a name and without a return type.
- If return type is omited, it is actually deduced from the body's return statement.
- The *capture* is a list of variables that should be shared with the function's body. These do not have to be global variables!
- So our code can become:

```
auto a = new double [5] {2.0,1.1,1.2,1.3,1.4};
double sum {0.0};
for_each(a,a+5, [&sum](double x){ sum+=x; });
```

- Note the ampersand in the capture.

SciNet compute • calcul CANADA

Library enhancements

- A lot of functionality has been added to C++11 in the form of standard libraries. This includes stuff that was in TR1.
- Many new features as well. Will pick a few.
- Intel compiler depends on underlying g++ for many of these. Need both g++ 4.7.2 and icpc 13.1 for some of this to work.

Library enhancements

In particular let's consider:

- Smart pointers
- Random numbers
- Timing routines
- Threading

Smart pointers

- Smart pointers help avoid memory issues like
  1. Not deallocating `new`-ed memory because `delete` is missing.
  2. Not deallocating `new`-ed memory because of an exception.
  3. Passing a pointer to a class that may use it after its lifetime.
  4. Or that may try to delete it.
- Three useful types:
  1. `std::unique_ptr`
     A pointer wrapper that will deallocate the memory associated with the pointer when it goes out of scope. Cannot be copied. Can use `*` and `->`.
  2. `std::shared_ptr`
     A pointer wrapper that will deallocate the memory associated when there are no more reference to it. Internally increases a reference counter when copied.
  3. `std::weak_ptr`
     Like `std::shared_ptr` but without increasing the reference count. Sometimes useful, but rare.
  4. `std::auto_ptr`
     Deprecated.
- In the header file `<memory>`.

Random numbers

- The header file `<random>` has random number generators
- Unlike previous compiler RNGs, these are actually good.
- They're extensible too.
- The random library separates the random number generator from the distribution that numbers are to be drawn from.

```
#include <random>
#include <functional>
int main()
{
   std::uniform_int_distribution<int> distribution(0,99);
   std::mt19937 engine; // Mersenne twister MT19937
   auto generator = std::bind(distribution, engine);
   engine.seed(13);
   int random1 = generator();
   int random2 = distribution(engine);
}
```

Timing routines

- The header file `<chrono>` has standard timing routines.
- Comes with a large number of templates to describe time units, durations, etc.

Example

```
#include <chrono>
int main()
{
   using namespace std::chrono;
   for (int i = 0; i < 10; i++) {
      auto tick = steady_clock::now();
      // do something
      auto tock =steady_clock::now();
      float time = duration<float>(tock-tick).count();
      std::cout << time << "s" << std::endl;
   }
}
```

Multi-threading

- Threads are part of the standard i.e. not an added on library.
- Atomic data type in the standard.
- The header file `<thread>` implements the thread class.
- The header file `<future>` implements asynchronous stuff and references to not-yet computed values.
- Very reminiscent of pthreads, but much easier to program for simple cases, while maintaining flexability.
- Too large a subject to properly explain here, let's look at some examples.

Multi-threading example 1

```cpp
#include <iostream>
#include <thread>
void threadfunction() {
   std::cout << "Hello from thread!\n";
}
int main() {
   std::thread th(&threadfunction);
   std::cout << "Hello world!\n";
   th.join();
   return 0;
}
```

Multi-threading example 2

```
#include <iostream>
#include <vector>
#include <thread>
int main()
{
   std::vector<int> a {1,2,3};
   std::cout << "a.size() = " << a.size() << std::endl;
   std::thread th1([]() {
      std::cout << "Hello from thread!" << std::endl;
   });
   std::thread th2(&std::vector<int>::clear,std::ref(a));
   std::cout << "a.size() = " << a.size() << std::endl;
   std::cout << "Hello world!" << std::endl;
   th1.join();
   th2.join();
   std::cout << "a.size() = " << a.size() << std::endl;
   return 0;
}
```

Note: for icpc, this lambda in a thread only works when wrapped in a `std::function<void(void)>`.

References

`http://wiki.apache.org/stdcxx/C++0xCompilerSupport`
`http://www.cplusplus.com/reference`
`http://en.cppreference.com/w/`