C++11: What's New?

Jason Merrill

GNU C++ Maintainer

Red Hat, Inc.

constexpr

```
struct A {
  int i;
  constexpr A(int _i): i(_i) {}
  constexpr operator int()
    { return i; };
};
constexpr A fttw(42);

int ar[fttw];
```

variadic templates

- Variadic templates rely on recursion

```
template<class...> struct Tuple;
template<class T,class...Rest>
 struct Tuple<T, Rest...>:
  public Tuple<Rest...>
 { T t; };
template<> struct Tuple<> {};
```

variadic templates

```
template<class T, class...Rest>
void f(T t, Rest... rest)
{
  frob (t);
  f(rest...);
}
void f() { }
```

threads

- memory model (also adopted by C11)
  - “happens before”
  - data races are undefined behavior
- atomics library
  - memory order: relaxed, release, acquire, consume, seq_cst
- threads library
  - threads, mutexes, condition vars, futures

auto and decltype

- `auto x = f(y);`
- `decltype(f(y)) x;`
- `x = f(y);`

decltype vs. decltype

```
int i;
struct A {
  int i;
} a;

// int
decltype(i) j;
decltype(a.i) k;
```

```
int i;
struct A {
  int i;
} a;

// int&
decltype((i)) j;
decltype((a.i)) k;
```

decltype in return type

- ```
  template<class T, class U>
    decltype(*(T*)0+*(U*)0)
  add(T t, U u)
  { return t + u; }
  ```
  ```

decltype in return type

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

enhanced SFINAE

- Substitution Failure Is Not An Error
- If `t+u` is ill-formed for some T and U, template argument deduction fails.
- ```
  template<class T, class U>
  auto add(T t, U u)
     -> decltype (t + u)
  { return t + u; }
  ```
  ```

enhanced SFINAE

- Use the `enable_if` template to test for particular conditions
- ```
  template <class T>
  typename enable_if
   <is_scalar<T>::value, T>
   ::type
  f(T); // version for scalars
  ```
  ```

enhanced SFINAE

- Can also use SFINAE in default template arguments

```
template <class T, class
 = typename enable_if
    <is_trivial<T>::value,
     void>::type>
A(T);
```

- But be careful with overloading

lvalues and rvalues

- lvalue (“left”):
  has object identity,
  can have its address taken,
  can have dynamic type different from static type.
- rvalue (“right”):
  can be copied freely,
  might not live in memory,
  has fixed type.

rvalue references

- ```
  template<class T>
  void swap(T& a, T& b) {
    T t (a);
    a = b;
    b = t;
  }
  ```
- Three expensive copies
  ```

rvalue references

- Add move constructors and assignment ops

```
struct A {
  Rep* rep;
  A& operator=(const A&);
  A& operator=(A&& rhs) {
    rep = rhs.rep;
    rhs.rep = nullptr;
  }
};
```

rvalue references

- ```
  A f();
  extern A a;
  a = f(); // move from rvalue
  ```

- ```
  template<class T>
  void swap(T& a, T& b) {
    T t (move (a)); // force rvalue
    a = move(b);
    b = move(t);
  }
  ```

perfect forwarding

- Old forwarding function:

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

- Value category (lvalue/rvalue) of the arguments to `add` is lost.

perfect forwarding

- Now:

```
template <class T, class U>
auto add(T&& t, U&& u)
   -> decltype (t + u)
{ return forward<T>(t)
         +forward<U>(u); }
```

- Value category is retained.

xvalues

- lvalue: identity, address, polymorphism, not movable
- prvalue: free copying, no address, fixed type, movable
- xvalue (“expiring”): identity, address, polymorphism, movable
- glvalue: identity, address (lvalue or xvalue)
- rvalue: movable (prvalue or xvalue)

noexcept

- What if we throw in the middle of a move?

```
template<class T>
void swap(T& a, T& b) {
  T t (move (a));
  a = move(b); // THROWS
  b = move(t);
}
```

- Data loss

noexcept expression

- Test whether an expression can throw

```
template<class T>
void swap(T& a, T& b) {
  static_assert(
    noexcept(a = move(b)),
    “operator= could throw”);
  T t (move (a));
  a = move(b);
  b = move(t);
}
```

noexcept specifier

- Mark move ctor and op= as non-throwing

```
struct A {
  Rep* rep;
  A& operator=(const A&);
  A& operator=(A&& rhs) noexcept
  {
    rep = rhs.rep;
    rhs.rep = nullptr;
  }
};
```

noexcept metaprogramming

```
template<class T>
void swap(T& a, T& b)
 noexcept(noexcept(T(move(a))) &&
          noexcept(a = move(b)))
{
  T t (move (a));
  a = move(b);
  b = move(t);
}
```

noexcept metaprogramming

```
template<class T>
void swap(T& a, T& b)
noexcept(__and_
 <is_nothrow_move_constructible<T>,
  is_nothrow_move_assignable<T>
  >::value)
{
  T t (move (a));
  a = move(b);
  b = move(t);
}
```

initializer lists

- Previously:

```
std::map<char,int> m;
m['a'] = 42;
m['b'] = 237;
....
m.insert(pair<char,int>
            ('c',1024));
```

initializer lists

- Now:

```
std::map<char,int> m = {
  { 'a', 42 },
  { 'b', 237 },
  /* ... */
};
m.insert({'c',1024});
```

narrowing

```
char c1 { 1024 };   // error
char c2 { 42 };     // OK
char c3 { c2 + 1 }; // error :(
```

range-based for

- Previously:

```
T::iterator it;
for (it = t.begin();
     it != t.end;
     ++it)
  {
    T::iterator::value_type&
      v = *it;
    frob (v);
  }
```

range-based for

- Now:

```
for (auto& v : t)
  frob (v);
```

- Works with C arrays and any class with begin/end member functions that return iterators.

lambda

- Previously:

```
struct AddN {
 int n;
 AddN(int _n): n(_n) {}
 operator()(int& x) { x+=n; }
};
...
AddN addi(i);
```

lambda

- Now:

```
auto addi =
  [=](int& x){ x+=i; };
```

alias templates

- Previously:

```
template <class T>
struct ptr {
  typedef T* type;
};

template <class T> void
f(typename ptr<T>::type);
```

- Not deducible

alias templates

- Now:

```
template<class T>
using ptr = T*;

template<class T>
void f(ptr<T>);
```

- Deducible (and shorter)

non-static data member initializers

- Previously:

```
struct A {
  int i;
  A(): i(42) { }
  A(something): i(42) { ... }
};
```

non-static data member initializers

- Now:

```
struct A {
  int i = 42;
};
```

- If a constructor does not have a mem-initializer for a particular non-static data member, it uses the NSDMI instead.

delegating constructors

- Alternately:

```
struct A {
  int i;
  A(): i(42) { }
  A(something): A() { ... }
};
```

inheriting constructors

- Previously:

```
struct A {
  A(int);
  A(const char *);
};

struct B: A {
  B(int i): A(i) { }
  B(const char *p): A(p) { }
};
```

inheriting constructors

- Now:

```
struct A {
  A(int);
  A(const char *);
};

struct B {
  using A::A;
};
```

- Combines well with NSDMI

defaulted member functions

- Previously:

```
class A {
  int i, j;
 protected:
  A(const A& a):i(a.i),j(a.j){}
};
```

- Copy constructor is non-trivial

defaulted member functions

- Now:

```
class A {
  int i, j;
 protected:
  A(const A&) = default;
};
```

- Copy constructor is trivial
- Can use `= delete` to completely suppress implicitly defined functions

raw strings

```
string backslashes = “\\\\+”;
string raw_bslshes = R”(\\+)”;

string bsq = “\”\\(\\\\+\\)\””;
string rbq = R”#(“\(\\+\)”)#”
```

user-defined literals

```
// Poor approximation, yes

constexpr long double
 operator”” _degrees
  (long double d)
{ return d * 0.0175; }

constexpr long double pi
 = 180_degrees;
```

enum class

- Enumerator names require explicit scope
- No implicit conversion to integer type

```
enum E1 { a, b }; // traditional
if (a == 0) // OK


enum class E2 { a, b }; // OK
if (E2::a == 0) // type mismatch
```

opaque enum declaration

```
enum E: int;
E* ep;
enum E: int { e1, e2 };
```

unrestricted unions

```
union U {
   int i;
   string s;
};
U u; // error, U() is deleted
```

unrestricted unions

```
union U {
   int i;
   string s;
   U(): i() { }
   ~U() { }
};
U u; // OK
```

explicit conversion ops

```
template<class T> struct Ptr {
  // ...
  operator bool();
};

Ptr<int> ip;
if (ip) { /*...*/ } // good
Ptr<float> fp;
ip == fp; // bad, but well-formed
```

explicit conversion ops

```
template<class T> struct Ptr {
  // ...
  explicit operator bool();
};

Ptr<int> ip;
if (ip) { /*...*/ } // still OK
Ptr<float> fp;
ip == fp; // error, type mismatch
```

override

```
struct A {
  virtual void f();
};

struct B: public A {
  void f() override; // OK
  void f(int) override; // error
};
```

final

```
struct A {
  virtual void f() final;
};
struct B: public A {
  void f(); // error
};

struct C final { };
struct D: public C { }; // error
```

Libraries: Containers

- Singly-linked list: `forward_list`
- Hash tables: `unordered_set/map`
- N-dimensional “pair”: `tuple`
- Generalized array: `array`

Libraries: Other

- Time: `chrono`
- Rational arithmetic: `ratio`
- Generalized function: `function`
- Regular expressions: `basic_regex`

Future of C++

- Aiming for the next standard in 2017
- Mostly bug fixes, small improvements
- Probably one major new feature
  - Improved concurrency support?
  - Modules?
  - Reflection?

Any questions?