An overview of C++11, and C++14 changes (part 2/3)

Ricard Sierra Rebull

May 15th, 2019

Language changes (II)

- Keywords
- Declarations
- Initializations
- Lambda functions
- Templates
- Examples
- Compiler support

Language changes (II)

- **Keywords**
- Declarations
- Initializations
- Lambda functions
- Templates
- Examples
- Compiler support

New keywords (specific usage)

- **alignof**

```
struct S { int i, j, k; };
static_assert(alignof(S)==alignof(int), "weird alignment od struct S.");
```

- **alignas**

```
alignas(16) int i;
struct alignas(long) S1 { char a, b; };
struct alignas(alignof(long)) S2 { char a, b; };
```

- **decltype**

```
vector<MyType> v1;
decltype(v1)::value_type j;    // j is of type MyType
```

- **thread_local**

```
extern thread_local unsigned int count = 1;     // namespace scope and external linkage
static thread_local unsigned int count = 1;     // namespace scope and internal linkage
static thread_local unsigned int count = 1;     // static data member and external linkage
thread_local unsigned int count = 1;            // block scope, equivalent to static thread_local
```

Language changes (II)

- Keywords
- **Declarations**
- Initializations
- Lambda functions
- Templates
- Examples
- Compiler support

New declarations

- attributes

```
[[noreturn]] void f() { throw "error"; }
void g([[carries_dependency]] int i);              // fence optimization
struct [[deprecated("Error message")]] X { ... };  // valid since C++14
enum [[gnu::unused]] X { ... };                    // implementation defined
```

- alternate function declarations

```
auto f() -> int;
auto f() -> int(*)();
auto f() { return 0; }            // valid since C++14
auto f() -> auto { return 0; }    // valid since C++14
decltype(auto) f() { return 0; }  // valid since C++14

struct S {
  auto getChar() -> char;
  auto getShort() -> short;
  auto getInt() -> int;
  auto getLong() -> long;

  bool isOpen() const;
  int encoderValue() volatile;
  void reset() noexcept;
  Result log() &;
  Result log() &&;

  const decltype(auto) fullSyntax1() const volatile & noexcept;
  const decltype(auto) fullSyntax2() volatile const && noexcept;
};
```

Language changes (II)

- Keywords
- Declarations
- **Initializations**
- Lambda functions
- Templates
- Examples
- Compiler support

New initializations (I)

- value initialization: solve misinterpretation as function declaration (ADDED)

```
T object{};                              // named variable
```

- value initialization: like C++98 but replacing () by {}

```
T{};                                     // temporal value
new T{};                                 // dynamic storage value
Class::Class(...) : member{} ... {...}   // data member
```

- direct initialization: like C++98 but replacing () by {} (non-class types) (no narrowing conversion)

```
T obj{ arg };            // T obj( arg ); if T is a non-class type
```

- aggregate initialization: omit the '=' character (arrays and POD class types)

```
T obj{ arg, arg2, ... }; // T obj = { arg1, arg2, ... }; if T is array or POD
```

New initializations (II)

- reference initialization

```
T&& ref( obj );
T&& ref = obj;

// list initialization
T&& ref{ arg1, arg2, ... };
T&& ref = { arg1, arg2, ... };
```

- examples

```
int foo();
int n = 1;
int&& r1 = n;            // error, cannot bind to lvalue
int&& r2 = 1;            // ok, bind to rvalue
int&& r3 = foo();        // ok, bind to rvalue
double&& r4 = 1;         // ok, bind to temporary with value 1.0
double&& r5 = (double)n; // ok, bind to temporary with value 1.0

// lvalue references
int& rref = foo();       // error, cannot bind to rvalue
int const& cref = foo(); // ok, rvalue liftime is extended to cref lifetime

// WARNING: temporal values have lifetime of the expression
struct S {
  A& a_; S(A& a) : a_(a) {}
};
A a;
S s1 = a;                // ok
S s2 = A();              // error, cannot bind to lvalue but ok with S::S(A const&);
```

List initialization (I)

- **std::initializer_list<T>**: proxy object defined as array of const T
  - list initialization of an object
  - argument of function call with initializer_list parameter
  - braced init list is bound to **auto**
- forbids narrowing conversions
- allow containers initialization to be defined as a C array
- have priority over other initializations

List initialization (II)

```
// named object
T obj{v1, ..., vn}; T obj = {v1, ..., vn};

// data member
Class { T obj{v1, ..., vn}; }; Class { T obj = {v1, ..., vn}; };
Class::Class(...) : member{v1, ..., vn} {...}

// temporary object
T{v1, ..., vn}

// dynamic object
new T{v1, ..., vn}

// function parameter
foo( {v1, ..., vn} ); obj[ {v1, ..., vn} ]; obj = {v1, ..., vn};

// cast operator
U( {v1, ..., vn} )

// return object
return {v1, ..., vn};
```

List initialization (III)

```cpp
#include <initializer_list>
template<typename T> struct Container {
  Container(std::initializer_list<T>) { ... }
};
Container<int> c = {1,2,3,4,5};

// Equivalent to:
const int __temp[] = {1,2,3,4,5};
Container<int> c = Container<int>( {begin(__temp), end(__temp)} );
// or
template<typename T, size_t N> constexpr size_t size(T const (&)[N]) { return N; }
Container<int> c = Container<int>( {begin(__temp), size(__temp)} );
```

Language changes (II)

- Keywords
- Declarations
- Initializations
- **Lambda functions**
- Templates
- Examples
- Compiler support

Lambdas: Inline function objects (I)

```
[]{}    // Most simple lambda function

[ ... ]( ... ) mutable noexcept -> void { ... }
```

- The returned type is **auto** except explicitly declared
- Capture variables in scope:
  - explicit: [var1, &var2, this, ...]
  - explicit initialized(C++14): [v1=i+1, &v2=var2, v3=std::move(p), ...]
  - implicit (default byVal): [=], [=, &var1...]
  - implicit (default byRef): [&], [&, var1,...]
- By default captured values are const except set to **mutable**

```
// A mutable lambda can modify its captured values
int v = 0;
auto f = [v]() mutable -> int { return ++v; }
f();    // returns 1
f();    // returns 2
```

- Parameters must have specific type (C++11), or **auto** (C++14)

Lambdas: Inline function objects (II)

```cpp
// Captures by value, and b by reference
int a = 1, b = 2;
auto f = [a,&b](int i){ ++b; return a + b * i; };
int res = f(10);

// Equivalent code:
int a = 1, b = 2;

class __Anon {
  const int a;
  int& b;
public:
  __Anon(int _a, int const& _b) : a{_a}, b{_b} {}
  int operator()(int i) const { ++b; return a + b * i; }
};
__Anon f{a,b};

int res = f(10);
```

Lambdas: Inline function objects (II)

```cpp
// Captures by value, and b by reference
int a = 1, b = 2;
auto f = [a,&b](auto i, auto j){ ++b; return a + b - j * i; };
int res = f(10);

// Equivalent code:
int a = 1, b = 2;

class __Anon {
  const int a;
  int& b;
public:
  __Anon(int _a, int const& _b) : a{_a}, b{_b} {}

  template<typename T1, typename T2>
    int operator()(T1 i, T2 j) const { return a + b - j * i; }
};
__Anon f{a,b};

int res = f(10,11);
```

Language changes (II)

- Keywords
- Declarations
- Initializations
- Lambda functions
- **Templates**
- Examples
- Compiler support

Template changes (I)

- right angle brackets

```cpp
set<vector<Foo>> foos;
```

- extern template: compilation time optimization

```cpp
//+++ file: m1.c (no instantiation code generated)
#include <string>
extern template class std::basic_string<char>;

std::string f1() { return {}; }

//+++ file: m2.c (no instantiation code generated)
#include <string>
extern template class std::basic_string<char>;

void f2() { std::string s; ... }

//+++ file: main.c (template instance code generated)
#include <string>

// force instantiation of std::string
template class std::basic_string<char>; // optional, only needed if no std::string used

int main () {
  ...
}
```

- variable templates(C++14)

```cpp
template<typename T> T pi = 3.1415926535897932385L;          // only in namespace scope
template<typename T> static T pi = 3.1415926535897932385L;   // in class scope must be static
double area = pi<double> * r*r ;
```

Template changes (II)

- local and unnamed types as template parameters

```
template<class T> class X {};
template<class T> void f(T) {}

struct {} unnamed_obj;

void foo() {
  f(unnamed_obj);        // ok, unnamed type
  struct A {}; X<A> x1; // ok, local type
  enum { e1 }; f(e1);   // ok, local unnamed type
}
```

- variadic templates: parameter pack

```
// template function
template<typename... T> void foo (T... args);
foo();
foo(1, 'a', 2.5);

// template class
template<typename... T> struct X {};
X<> x1;
X<int,char> x2;
```

Template changes (III)

- parameter pack expansion

```
// sizeof... operator
sizeof...(args)      // number of args

// function parameter list
template<typename... T> void f(T...);
// function argument list
f(n, ++args...)

// template parameter list
template<typename T, typename... Ts> class X;
// template argument list
X<int,Ts...> x;

// initializers
X x(std::forward<Args>(args)...);
int table[sizeof...(args)] = { (cout << args,0)... };

// base specifiers and member initializer
template<typename... Ts> struct V : Ts... {
  X(Ts const&... elems) : Ts(elems)... {}
}

// lambda captures
auto res = [&, args...]{ return f((args + 2)...); }();

// alignas
alignas(args...) char buff[256];
```

Language changes (II)

- Declarations
- Initializations
- Lambda functions
- Templates
- **Examples**
- Compiler support

Unrestricted unions example

```
union S {
  std::string str;
  std::vector<int> vec;
  ~S() {}
};

S s = {"Hello, world"};
...
s.str.~basic_string();
new (&s.vec) std::vector<int>;
```

Should be:

```
union S {
  std::string str;
  std::vector<int> vec;
  ~S() {}
};

S s = {"Hello, world"};
...
std::string::allocator_type str_alloc;
std::allocator_traits<std::string::allocator_type>::destroy(str_alloc, &s.str);

std::vector<int>::allocator_type vint_alloc;
std::allocator_traits<std::vector<int>::allocator_type>::construct(vint_alloc, &s.vec);
```

User defined literals

- user-defined literals must start with '_'
- default arguments are not allowed

character literal

```
operator""_xxx(char)
operator""_xxx(wchar_t)
operator""_xxx(char16_t)
operator""_xxx(char32_t)
```

string literal

```
operator""_xxx(const char *, size_t)
operator""_xxx(const wchar_t *, size_t)
operator""_xxx(const char16_t *, size_t)
operator""_xxx(const char32_t *, size_t)
```

integer literal

```
operator""_xxx(unsigned long long)
operator""_xxx(const char *)
template<char...> operator""_xxx()    // evaluated at compile-time
```

floating-point literal

```
operator""_xxx(long double)
operator""_xxx(const char *)
template<char...> operator""_xxx()    // evaluated at compile-time
```

User defined literals examples

Examples

```
// --- std::string literals ---
string operator ""s(const char *str, std::size_t len) {
  return std::string(str, len);
}
auto s = "This is a std::string"s;

// --- binary literals ---
template <unsigned VAL>
constexpr unsigned build_binary_literal() { return VAL; }

template <unsigned VAL, char DIGIT, char... REST>
constexpr unsigned build_binary_literal() {
  return build_binary_literal<(2 * VAL + DIGIT - '0'), REST...>();
}

template <char... STR>
constexpr unsigned operator""_b()
{
  return build_binary_literal<0, STR...>();
}

int n = 1011_b;     // n = 11
```

Range for

Equivalent to:

```
{
  auto && __range = range_expression ;
  for (auto __begin = begin_expr, __end = end_expr; __begin != __end; ++__begin) {
    range_declaration = *__begin;
    loop_statement
  }
}
```

Needs:

- range
  - C-array: nothing
  - Class type: begin(), end() members
  - Otherwise: begin(), end() functions
- begin()/end() returned type:
  - preincrement operator
  - indirection operator
  - inequality operator

Range for example (OOP)

```
class range final
{
public:
    class const_iterator final
    {
    public:
        constexpr const_iterator(int from, int to, int step) noexcept:
          count_{from}, end_{to}, step_{step} {}

        auto operator++() noexcept -> const_iterator&
          { count_ += step_; return *this; }
        constexpr auto operator*() const noexcept -> int
          { return count_; }
        constexpr auto operator!=(const_iterator it_end) const noexcept -> bool
          { return (step_ < 0) ? (it_end.end_ < count_) : (count_ < it_end.end_); }
    private:
        int count_{0};
        const int end_{0};
        const int step_{1};
    };

    constexpr explicit range(int to) noexcept: range{0, to} {}
    constexpr range(int from, int to, int step = 1) noexcept:
      from_{from}, to_{to}, step_{step} {}

    constexpr auto begin() const noexcept -> const_iterator
      { return const_iterator{from_, to_, step_}; }
    constexpr auto end() const noexcept -> const_iterator
      { return const_iterator{from_, to_, step_}; }

private:
    const int from_{0}, to_{0}, step_{1};
};
```

Range for example (functional)

```cpp
struct range final
{
    const int from{0}, to{0}, step{1};

    struct const_iterator;
    constexpr explicit range(int to) noexcept: range{0, to} {}
    constexpr range(int from, int to, int step = 1) noexcept: from{from}, to{to}, step{step} {}
};
struct range::const_iterator final
{
#if __cplusplus >= 201406L
    int count{0}; const int end{0}, step{1};
#else
    int count; const int end, step;
#endif
};

constexpr auto begin(range r) noexcept -> range::const_iterator
  { return range::const_iterator{r.from,r.to,r.step}; }
constexpr auto end(range r) noexcept -> range::const_iterator
  { return range::const_iterator{r.from,r.to,r.step}; }

constexpr auto operator*(range::const_iterator it) noexcept -> int
  { return it.count; }
#if __cplusplus >= 201406L
  constexpr auto operator++(range::const_iterator& it) noexcept -> range::const_iterator&
#else
  inline auto operator++(range::const_iterator& it) noexcept -> range::const_iterator&
#endif
  { it.count += it.step; return it; }

constexpr auto operator!=(range::const_iterator it, range::const_iterator it_end) noexcept -> bool
  { return (it.step < 0)? (it_end.end < it.count) : (it.count < it_end.end); }
```

Variadic templates

```
// Variadic function
template<typename T>
  int adder(T val)
  { return val; }
template<typename T, typename... Args>
  int adder(T val, Args... args)
  { return val + adder(args...); }

// Variadic type
template<size_t idx, typename T>
  struct TupleElem { T value };

template<size_t, typename...>
  struct TupleImpl;
template<size_t N>
  struct TupleImpl<N> {};
template<size_t idx, typename T, typename... TRest>
  struct TupleImpl<idx,T,TRest...> :
    TupleElem<idx,T>,
    TupleImpl<idx+1,TRest...> {};

template<typename... T>
  using Tuple = TupleImpl<0,T...>;

template<size_t idx, typename T, typename... TRest>
  T& get (TupleImpl<idx,T,TRest...>& t)
  { return t.TupleElem<idx,T>::value; }

template<size_t idx, typename... T>
  size_t num_elem (TupleImpl<idx>& t)
  { return idx; }
```

Errata

New C++14 features:

- Binary literals: 0b10011001
- Digit separators: 1’000’000, 1’2’3.00, 0xffff’ffff

END