Range-based for, nullptr, Enums, Literals, static_assert, noexcept

C++11 Features in GCC 4.8

pluralsight
hardcore developer training

Overview

- Range-based for loop
- nullptr
- Enum changes
- Unicode support, raw string literals, user defined literals
- Compile time assertions with static_assert
- Exception specifications with noexcept

Range-based for Loop

```
vector<int> v;
// ... populate the vector ...
for (auto elem : v)
    cout << elem << endl;

for (auto& elem : v)
    elem *= 2;


for (int elem : v)
    cout << elem << endl;


int arr[] = {10, 20, 30, 40};
for (auto elem : arr)
    cout << elem << endl;
```

Looping over initializer_list

```
auto list = {100, 200, 300, 400};
for (auto elem : list)
    cout << elem << endl;
```

Range-based Looping For Your Class

```
class MyContainer
{
    list<int> _values {111, 222, 333};
public:
    friend list<int>::iterator begin(MyContainer& cont);
    friend list<int>::iterator end(MyContainer& cont);
    // ...
};

list<int>::iterator begin(MyContainer& cont)
{
    return cont._values.begin();
}

list<int>::iterator end(MyContainer& cont)
{
    return cont._values.end();
}


MyContainer cont;
for (auto& elem : cont)
    cout << elem << endl;
```

Range-based for Loop Internals

```
for (elem_decl : seq)
    statement;
```

```
for (auto iter = seq.begin(), seq_end = seq.end(); iter != seq_end; ++iter)
{
    elem_decl = *iter;
    statement;
}
```

```
for (auto iter = begin(seq), seq_end = end(seq); iter != seq_end; ++iter)
{
    elem_decl = *iter;
    statement;
}
```

nullptr

```
int* p = nullptr;
```

```
namespace std
{
   typedef decltype(nullptr) nullptr_t;
}
```

```
int* p = nullptr;
int* p1 = NULL;
int* p2 = 0;
p1 == p;    // true
p2 == p;    // true
```

```
int* p {};  // p is set to nullptr
```

How Is nullptr Better Than NULL?

```
bool ambiguous(int)
{
    return false;
}

bool ambiguous(int*)
{
    return true;
}

ambiguous(NULL);        // returns false, ambiguous(int) overload chosen

ambiguous(nullptr);     // returns true, ambiguous(int*) overload chosen
```

Enum Changes

- Strongly typed enums
- Forward declarations for enums
- Scoped enums

Scoped Enums

```
enum class Proportion
{
    OneHalf,
    OneThird,
    OneQuarter
};

Proportion prop = OneThird;                 // error

Proportion prop2 = Proportion::OneThird;    // OK
```

```
auto prop = Proportion::OneThird;
if (prop == 1)      // error
    // ...
```

Specifying the Underlying Type

```
enum Direction : unsigned short
{
    South,
    West,
    East,
    North
};

cout << sizeof(North) << endl;  // outputs sizeof(unsigned short)
```

```
enum Color : double             // error
{
    Black
};
```

Forward Declaration

```
// flight_board.h
enum class AirportCode; // forward declared enum

struct FlightBoard
{
    void print_airport_name(AirportCode code)
    {}

    void print_flight(AirportCode code, const string& flight)
    {
        // ...
        print_airport_name(code);
    }
};
```

Forward Declaration

```
// navigator.h
struct Navigator
{
    Navigator();
private:
    enum CompassPoint : int;  // forward declaration
    CompassPoint _compass_point;
};

// navigator.cpp
enum Navigator::CompassPoint : int { North, South, East, West };

Navigator::Navigator() : _compass_point(North)
{}
```

Forward Declaration Rules

- Forward declaration has to include the type (implicitly or explicitly)
- The underlying type has to match between all declarations and definition
- Declarations can’t change from scoped to unscoped enum, or vice versa

Forward Declaration Examples

```
enum E : short;        // OK
enum F;                // error, underlying type is required
enum class G : short; // OK
enum class H;          // OK, underlying type for scoped enums is int by default

enum E : short;        // OK, redeclaration
enum class G : short; // OK, redeclaration
enum class H;          // OK, redeclaration
enum class H : int;    // OK, redeclaration with the same underlying type

enum class E : short; // error, can't change from unscoped to scoped
enum G : short;        // error, can't change from scoped to unscoped

enum E : int;          // error, different underlying type
enum class G;          // error, different underlying type
enum class H : short; // error, different underlying type

enum class H {};       // OK, this redeclaration is a definition
```

Compile Time Assertions

- Preconditions on template type parameters
- Validate non-type template parameters
- Enforce requirements for type sizes

static_assert

```
int int_magic(int a, int b)
{
    static_assert(sizeof(int) <= 4, "int must be no more than 4 bytes");
    // ... do things with a and b
}


template<unsigned int dimensions>
struct Matrix
{
    Matrix()
    {
        static_assert(dimensions <= 3, "dimensions must not exceed 3");
    }
};

Matrix<3> m3;     // OK

Matrix<4> m4;     // error
```

static_assert

```
struct Base
{
    virtual ~Base() {}
};

template<typename T>
class Derived : public T
{
    static_assert(has_virtual_destructor<T>::value,
        "The base class must have a virtual destructor");
};

Derived<Base> d;        // OK

Derived<string> s;      // triggers static_assert
```

Literals

- Unicode literals
- Raw literals
- User defined literals

Unicode Support and String Literals

```
u8"UTF-8: \u00BD"

u"UTF-16: \uA654"

U"UTF-32: \U0002387F"
```

Prefix	Character Type	String Type
u8	char	string
u	char16_t	u16string
U	char32_t	u32string

```
string s(u8"\u00BD \u00B5s");   // the string represents ½ µs
```

ISO/IEC 10646

u'\uA654' <==> U'\U0000A654'

Unicode Character Literals

Prefix	Character Type	Example Literal
u	char16_t	u'\u160E'
U	char32_t	U'\U0000160E'

Raw Literals

```
cout << R"(use "\n" for newlines)" << endl;
```

```
  R"No newline \n"
 LR"No newline \n"
u8R"No newline \n"
 uR"No newline \n"
 UR"No newline \n"
```

Raw Literals

```
R"("\w+\\w+")"
```

“ab\cd”

```
"\"\\w+\\\\w+\""
```

```
R"(grep -r "\.js" *)"
```

```
cout << R"!!(A raw literal is delimited with "( )")!!" << endl;
```

```
R"(multiline
literal)"
```

<==>

```
"multiline\nliteral"
```

User Defined Literals

```
1.2_i;  // express complex numbers
10_km;  // express units


widget.set_height(150_px);
widget.set_width(80_percent);


// create a 'complex' instance from an imaginary literal
constexpr complex<double> operator "" _i(long double d)
{
    return {0, d};
}
```

User Defined Literals

Integer literals

```
operator "" _suffix(unsigned long long)
operator "" _suffix(const char*)

template<char... Digits>
operator "" _suffix()
```

Floating point literals

```
operator "" _suffix(long double)
operator "" _suffix(const char*)

template<char... Digits>
operator "" _suffix()
```

Character literals

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

String literals

```
operator "" _suffix(const char*)
operator "" _suffix(const wchar_t*)
operator "" _suffix(const char16_t*)
operator "" _suffix(const char32_t*)
```

Literal Operators for Integers

```
constexpr Distance operator "" _au(unsigned long long n)
{
    return Distance(n, Unit::astronomical_unit);
}

constexpr double radius = (30_au).to_light_years();
cout << "Neptune orbit radius: " << radius << " light years" << endl;
```

Literal Operators for Integers

```
unsigned long long operator"" _b(const char* digits)
{
    if (strlen(digits) > numeric_limits<unsigned long long>::digits)
        throw runtime_error("Too many digits in binary literal");

    unsigned long long res = 0;
    auto digit = digits;
    while (*digit != '\0')
    {
        if (*digit != '1' && *digit != '0')
            throw runtime_error("Only 1 and 0 allowed in binary literals");

        res = (*digit - '0') + (res << 1);
        ++digit;
    }
    return res;
}


101_b;      // equals 5
-1011_b;    // equals -11


123_b;      // throws
```

Literal Operators for Integers

```
template<char... Digits>
constexpr unsigned long long operator "" _b()
{
    return Binary<Digits...>::value;
}
```

Literal Operators for Integers

```
template<char... Digits>
struct Binary;

template<char digit, char... Digits>
struct Binary<digit, Digits...>
{
    static_assert(
        sizeof...(Digits) + 1 <= numeric_limits<unsigned long long>::digits,
            "Too many digits in binary literal");
    static_assert(digit == '1' || digit == '0',
        "Only 1 and 0 allowed in binary literals");

    static constexpr unsigned long long value =
        ((digit - '0') << sizeof...(Digits)) + Binary<Digits...>::value;
};

template<>
struct Binary<>
{
    static constexpr unsigned long long value = 0;
};


constexpr auto value = 101_b;
```

Character and String Literal Operators

Character literals

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

String literals

```
operator "" _suffix(const char*)
operator "" _suffix(const wchar_t*)
operator "" _suffix(const char16_t*)
operator "" _suffix(const char32_t*)
```

Character and String Literal Operators

```
u16string operator "" _reverse(const char16_t* str, size_t len)
{
    u16string s(str);
    reverse(s.begin(), s.end());
    return s;
}


uR"(two\nlines)"_reverse;// yields "seniln\owt"


u"The quick brown fox "
"jumps over a lazy dog "
"and back in reverse"_reverse
```

```
u"The quick brown fox jumps over a lazy dog and back in reverse"_reverse
```

noexcept

- Byproduct of the introduction of move semantics
- noexcept specifier means function should not throw
- Dynamic exception specifications are deprecated
- Compiler generated functions are noexcept if all the operations they directly invoke are noexcept
- delete operators and user defined destructors are noexcept unless explicitly specified otherwise

noexcept for Your Own Functions

```
constexpr long fibonacci(int n) noexcept
{
    return n < 1 ? -1 :
        (n == 1 || n == 2 ? 1 : fibonacci(n - 1) + fibonacci(n - 2));
};
```

std::terminate

```
template<typename T>
auto square(const T& v) noexcept(is_fundamental<T>::value) -> decltype(v * v)
{
    return v * v;
}
```

```
~A() noexcept(false);
```

noexcept Operator

```
template<typename T>
auto square(const T& v) noexcept(noexcept(v * v)) -> decltype(v * v)
{
    return v * v;
}
```

- The operand isn't evaluated
- The analysis is limited to checking that all operations are noexcept

When to Use noexcept

- You are not likely to use it a lot; the default is to allow functions to throw
- Prefer to use it with small functions which are easy to analyze
- Avoid noexcept if your function has preconditions
- Move constructors and move assignment operators should be noexcept if possible

A Couple More Notes

```
long (*p_fib)(int) = fibonacci;  // OK, but noexcept is lost

long (*p_fib2)(int) noexcept = fibonacci; // instead, noexcept can be preserved


using Func = void(const string&) noexcept;
```

Summary

- Range-based for loop
- nullptr
- enum features
- Unicode support and new literals
- Compile time assertions and exception specifications