# The Rest of Language Features, Other Platforms, and the Future of C++

C++11 Features in GCC 4.8

pluralsight
hardcore developer training

# Overview

- Explicit conversion operators
- Inline namespaces
- Alignment keywords
- New capability of sizeof
- New memory model
- Thread local storage

# Overview

- Generalized attributes
- Updated definition of POD types
- Changes to unions
- Compatibility with C99
- Deprecated and removed features
- C++11 support in other compilers
- Features planned for C++14

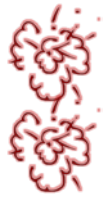
# Explicit Conversion Operators

```
explicit operator bool() const noexcept;
```

```
function<void(int, int)> f1, f2;

auto sum = f1 + f2;  // error

bool flag = f1;      // error
```



```
if (f1)
    f1(10, 20);
```

# Inline Namespaces

```
namespace API
{
    inline namespace v2
    {
        // v2 processes doubles instead of ints
        void process(vector<double>)
        {}
    }

    namespace v1
    {
        void process(vector<int>)
        {}
    }
}


vector<double> doubles;
API::process(doubles);

vector<int> ints;
API::v1::process(ints);
```

# Why Not Just Add a using Statement?

```
// library header
namespace API
{
    namespace v2
    {
        // v2 processes doubles instead of ints
        void process(vector<double>)
        {}
    }
    using namespace v2;
}
```

```
// user's code
namespace API
{
    // doesn't compile
    template<>
    class Hash<Part>
    {
        size_t operator()(const Part& p) const
        { /* ... */ }
    };
}
```

# alignof

```
alignof(double);  // yields 8
```

alignof(T&) <=> alignof(T)

alignof(T[N]) <=> alignof(T)

# alignas

```
alignas(32) int arr[10];

struct S
{
    alignas(32) Buffer _buf;
};

struct alignas(2 * alignof(double)) Doubled
{};
```

```
alignas(double) unsigned char double_buf[256];
```

```
alignas(T) alignas(A) T buffer[N];
```

# sizeof Applied to Non-static Data Members

```
class A
{
public:
    int _a;
};

sizeof(A::_a);      // yields sizeof(int)
```

# Memory Model

- Possibility of multi-threaded execution
- Defines compiler behavior vis-à-vis memory access
- Provisions for ordering memory operations and control over ordering

# Multi-threading Related Semantics

- Objects of static storage duration guaranteed to be initialized in a thread-safe manner
- Standard library requires const objects to be thread-safe
- mutable members of classes used with standard library must be synchronized internally

# Thread Local Storage

thread_local

__thread

# Thread Local Storage

```
thread_local B b1;

namespace
{
    thread_local B b2;
}

class C
{
    // each thread gets a separate instance counter
    static thread_local int _instance_counter;
};

void f()
{
    // each thread invoking f() has a copy of run_count
    static thread_local int run_count;
}
```

# Thread Local Storage

```
extern int i;
int thread_local i;           // error


extern int thread_local i;
int thread_local i;           // OK
```

# Thread Local Objects and Initialization

- Can be of types with arbitrary constructors and destructors
- Zero-initialized by default
- Can be initialized dynamically
- Namespace scope variables and static class members initialized before first use
- variables declared in a function are initialized similarly to local static variables
- An exception during construction results in a call to std::terminate

# Thread Local Destruction

- Destructors called in reverse order of construction but construction order is unspecified
- If a thread calls std::exit or exits from main, its variables are destroyed
- Variables on other threads are not destroyed

# Generalized Attributes

__attribute

__declspec

[[attribute_name]]

# Generalized Attributes

```
vector<int> v = {1, 2, 3, 4};
// ...
int a = v[[] { return 1; }()];   // error


int b = v[ [] { return 2; }()];  // error


auto lambda = [] { return 2; };
int c = v[lambda()];             // OK
```

# Standard Attributes

```
[[noreturn]] void thrower(const string& error) noexcept(false)
{
    throw runtime_error(error);
}
```

```
void f(int* p_handle [[carries_dependency]]);
```

```
struct [[gnu::aligned (32)]] S {};
```

```
struct __attribute((aligned (32))) S {};
```

# POD Types


- Trivial class
- Standard layout class
- All non-static data members are POD structs, POD unions or arrays of those

# Trivial Classes

- A scalar type
- A trivially copyable class with a trivial default constructor
- An array of one of those
- Optionally const- or volatile-qualified

# Trivially Copyable Classes

- No virtual functions or virtual base classes
- Trivial copy and move constructors
- Trivial copy and move assignment operators
- Trivial destructor

# Trivial Operations

- Not user-provided
- The containing class has no virtual functions or virtual base classes
- All the base class copy and move constructors are also trivial
- Copy and move constructors are trivial for all non-static data members of the class which are of class types or arrays of class types

# Standard Layout Types

- A scalar type
- Standard layout class
- An array of one of those
- Optionally const- or volatile-qualified

# Standard Layout Classes

- All non-static data members are standard layout (or arrays thereof)
- The same access control for all non-static data members
- No virtual functions or virtual base classes
- All base classes are standard layout too
- At most one class in the inheritance hierarchy has non-static data members
- No base classes of the same type as the first non-static data member

# Standard Layout Classes

```
struct Trivial          // trivial but not standard-layout
{
    int a;
private:
    int b;
};

struct StandardLayout   // standard-layout but not trivial
{
    int a;
    int b;
    ~StandardLayout();
};
```

# Changed Restrictions on Unions

- Members of types with user defined constructors, destructors and assignment operations are allowed
- Member types can't have virtual functions or to be reference types
- Operations of non-static data members can trigger removal of corresponding union operations

# Changed Restrictions on Unions

```
union Compact
{
    int _i;
    string _s;
};


Compact c1;   // error
```

```
union Compact
{
    int _i;
    string _s;

    Compact() : _i(100)
    {}
    ~Compact()
    {}
};

Compact c1;

cout << c1._i << endl;   // outputs 100
```

# Changed Restrictions on Unions

```
new (&c1._s) string("ABC");      //switch the active member to _s

// ... some time later
c1._s.~string();                 // destroy the string before making _i active
c1._i = 0;                       // switch the active member back to _i
```

# Discriminated Unions

```
class Compact
{
    enum
    {
        Int,
        String
    } _active_type;

    union
    {
        int _i;
        string _s;
    };
};
```

# Discriminated Unions

```
Compact(int i) : _active_type(Int), _i(i)
{}

Compact(const string& s) : _active_type(String)
{
    new (&_s) string(s);
}

~Compact()
{
    if (_active_type == String)
        _s.~string();
}
```

# Discriminated Unions

```
int get_int() const
{
    if (_active_type != Int)
        throw runtime_error("Inactive type requested");

    return _i;
}

void set(int i)
{
    if (_active_type == String)
        _s.~string();

    _i = i;
    _active_type = Int;
}
```

# Discriminated Unions

```
string get_string() const
{
    if (_active_type != String)
        throw runtime_error("Inactive type requested");

    return _s;
}

void set(const string& s)
{
    if(_active_type == String)
        _s = s;
    else
        new(&_s) string(s);
    _active_type = String;
}
```

# Discriminated Unions

```
Compact c1(100);

cout << c1.get_int() << endl;       // outputs 100

c1.get_string();  // throws because _s is inactive

string s("ABC");
c1.set(s);

cout << c1.get_string() << endl;    // outputs ABC
```

# C99 Compatibility Features

- long long type
- Standard C macros such as __func__ and __STDC_HOSTED__
- _Pragma(X) preprocessor operator
- Vararg macros and empty macro arguments
- Concatenation of wide and narrow strings

# Deprecated and Removed Features

- auto can no longer be used as a storage specifier
- export specifier for templates is no longer supported
- Dynamic exception specifications using throw are deprecated
- register storage class specifier is deprecated
- If the class has a user declared copy assignment operator or destructor, the compiler will not create a default copy constructor
- If the class has a user declared copy constructor or destructor, the compiler will not create a default copy assignment operator

# Writing Cross-platform Code

| | | |
|---|---|---|
| Clang |  | Full support |
| Intel |  | ✖ Thread local storage<br>✖ Changed restrictions on unions |
| Visual C++ |  | ✖ Many features partially supported or unsupported |

# Partially Supported Features in Visual C++

- Move semantics
- Defaulted functions
- thread_local isn't supported (but you can use __declspec(thread))
- C99 compatibility

# Unsupported Features in Visual C++

- constexpr
- Unicode support and literals
- User defined literals
- noexcept
- Inline namespaces
- Inheriting constructors
- Generalized attributes
- Alignment keywords
- sizeof with data members
- Arbitrary expressions in template deduction contexts
- Revised restrictions on unions

# C++14

`-std=c++1y`

# Return Type Deduction for Functions

```
auto square(int n)
{
    return n * n;
}
```

# Generic Lambdas

```
auto lambda = [](auto a, auto b) { return a * b; };


struct lambda1
{
    template<typename A, typename B>
    auto operator()(A a, B b) const
    {
        return a * b;
    }
};

auto lambda = lambda1();
```

# Extended Capturing in Lambdas

```
auto now = [val = system_clock::now()] { return now; };

now();  // returns current time


auto p = make_unique<int>(10);

auto lmb = [p = move(p)] { return *p; }
```

# Revised Restrictions on constexpr Functions

- Declarations, except static, thread_local or uninitialized variables
- if and switch statements
- Looping constructs, including range-based for
- Modification of objects contained within the constexpr expression

# constexpr Variable Templates

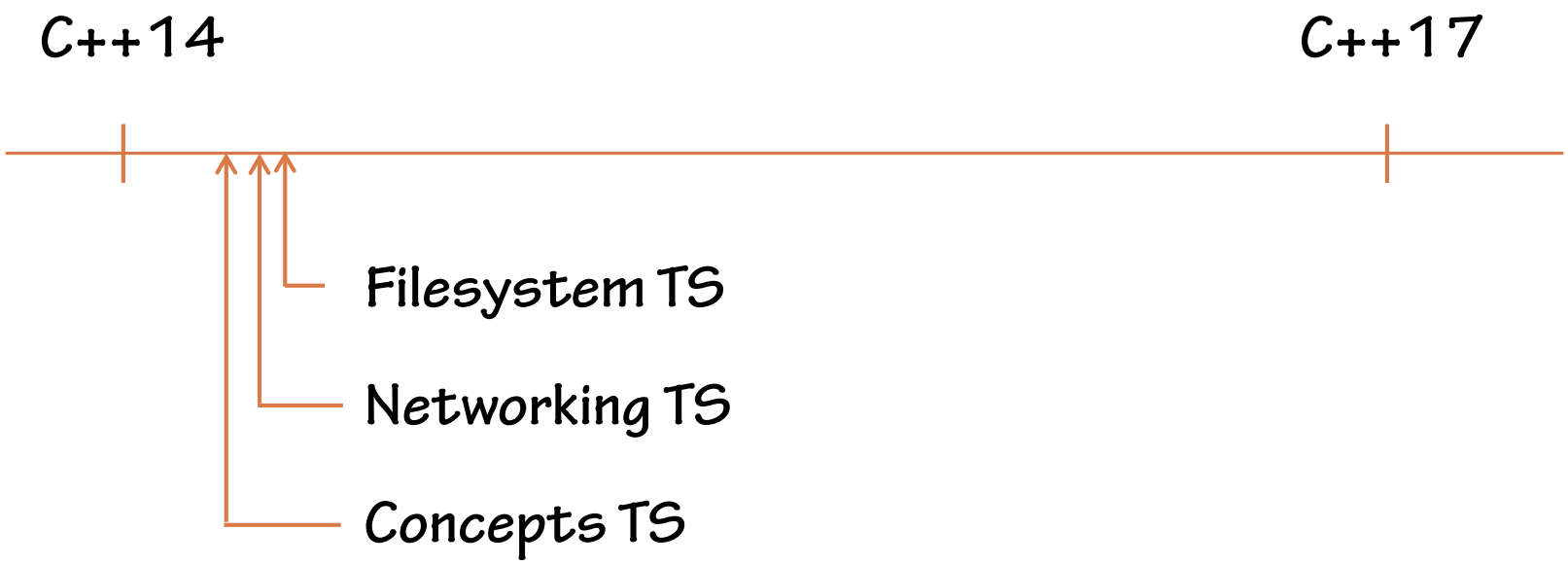
```
template<typename T>
constexpr auto T pi = T(3.14159265358979323385);


template<typename T>
T get_circle_area(T radius)
{
    return pi<T> * radius * radius;
}
```

# More Language Changes

- decltype(auto) for variable declarations
- Aggregate initialization combined with in-class initializers
- Runtime size for the last dimension of a stack allocated array
- Binary literals prefixed with 0b
- Separating digits with a quote: 1’000’000
- Standard attribute [[deprecated]]
- More standard literal suffixes: h, min, s, ms, us, ns

# Beyond C++14



# Summary

- Explicit conversion operators
- Inline namespaces
- Alignment keywords
- New capability of sizeof
- New memory model
- Thread local storage

# Summary

- Generalized attributes
- Updated definition of POD types
- Changes to unions
- Compatibility with C99
- Deprecated and removed features
- C++11 support in other compilers
- Features planned for C++14