Templates, Classes, Initialization

C++11 Features in GCC 4.8

pluralsight
hardcore developer training

Overview

- Template features
- Class features
- Overhauled syntax for initialization

Template Features Overview

- Variadic templates
- Template aliases
- Proper parsing of multiple closing angle brackets
- Local and unnamed types as template arguments
- extern templates
- Arbitrary expressions in template deduction contexts

Variadic Templates

```
function<bool(int, double)>
function<int(double, double, double)>
function<void(string, int, string, int, string, int)>
```

```
template<typename Stream, typename... Columns>
class CSVPrinter
{
public:
    void output_line(const Columns&... columns);
    // other methods, constructors etc. not shown
};
```

What Else Are Variadic Templates Good For?

- Perform type computation at compile time
- Generate type structure
- Implement type safe functions with arbitrary number of arguments
- Perform argument forwarding

Back to the Example

```
template<typename Stream, typename... Columns>
class CSVPrinter
{
public:
    void output_line(const Columns&... columns);
    // other methods, constructors etc. not shown
};
```

Working with Parameter Packs


```
void output_line(const Columns&... columns)
{
    write_line(validate(columns)...);
}
```

```
CSVPrinter<decltype(stream), int, double, string> printer;
```

```
void output_line(const int& col1, const double& col2, const string& col3)
{
    write_line(validate(col1), validate(col2), validate(col3));
}
```

Working with Parameter Packs

```
template<typename Value, typename... Values>
void write_line(const Value& val, const Values&... values) const
{
    write_column(val, _sep);
    write_line(values...);
}

template<typename Value>
void write_line(const Value &val) const
{
    write_column(val, "\n");
}


void output_line(const Columns&... columns);
```

Working with Parameter Packs

```
template<typename Stream, typename... Columns>
class CSVPrinter
{
    Stream& _stream;
    array<string, sizeof...(Columns)> _headers;
    // rest of implementation
};
```

Traversing Template Parameter Packs

```
template<typename... Types>                 // allow zero parameters
struct TupleSize;

template<typename Head, typename... Tail> // traverse types
struct TupleSize<Head, Tail...>
{
    static const size_t value = sizeof(Head) + TupleSize<Tail...>::value;
};

template<> struct TupleSize<>               // end recursion
{
    static const size_t value = 0;
};


TupleSize<>::value;                    // 0
TupleSize<int, double, char>::value;  // 13 on a 32-bit platform
```

Constraining Parameter Packs to One Type

```
template<typename... Strings>
void output_strings(const string& s, const Strings&... strings) const
{
    write_column(s, _sep);
    output_strings(strings...);
}

void output_strings(const string& s) const
{
    write_column(s, "\n");
}
```

More Places to Expand a Parameter Pack

```
template<typename... Bases>
class Derived : public Bases...
{};
```

Nested Variadic Templates

```
template<typename... Args1>
struct zip
{
    template<typename... Args2>
    struct with
    {
        typedef tuple<pair<Args1, Args2>...> type;
    };
};

typedef zip<short, int>::with<unsigned short, unsigned>::type T1;
// T1 is tuple<pair<short, unsigned short>, pair<int, unsigned>>

typedef zip<short>::with<unsigned short, unsigned>::type T2;
// error: different number of arguments specified for Args1 and Args2
```

One Function Template, Two Parameter Packs

```
template <size_t... Ns>
struct Indexes
{};

template<typename... Ts, size_t... Ns>
auto cherry_pick(const tuple<Ts...>& t, Indexes<Ns...>) ->
    decltype(make_tuple(get<Ns>(t)...))
{
    return make_tuple(get<Ns>(t)...);
}

// construct tuple<int, int, const char*, const char*, int, int>
auto data = make_tuple(10, 12012013, "B737", "Boeing 737", 2, 125000000);

// construct a tuple of (10, "B737", 2)
auto even_index_data = cherry_pick(data, Indexes<0, 2, 4>());
```

http://paulkeir.wordpress.com/2012/12/03/building-tuple-indices

Template Aliases

```
template<typename T>
using StrKeyMap = map<string, T>;
```

```
template<typename Stream>
struct StreamDeleter
{
    void operator()(Stream* os) const
    {
        os->close();
        delete os;
    }
};

template<typename Stream>
using StreamPtr = unique_ptr<Stream, StreamDeleter<Stream>>;
```

```
{
    StreamPtr<ofstream> p_log(new ofstream("log.log"));
    *p_log << "Log statement";
}  // stream gets closed and deleted here
```

Using using Instead of typedef

```
using PlaneID = int;

using Func = int(*)(double, double);
```

Closing Angle Brackets Are Allowed to Tail-Gate

```
vector<vector<int>> v;

map<int, vector<vector<int>>> m;
```

Local and Unnamed Types as Template Arguments

```
{
    struct A
    {
        string name() const { return "I'm A!"; }
    };

    vector<A> v(10);
    cout << v[0].name() << endl;
}
```

Local and Unnamed Types as Template Arguments

```
template <typename T>
void print(const T& t)
{
    t.print();
}

struct
{
    int x = 10;
    void print() const
    {
        cout << x;
    }
} a;

print(a);
```

extern Templates

```
// -- file1.h --
template<typename T>
T templated_func(const T& t)
{
    return t;
}

// -- file1.cpp --
using namespace std;

void f()
{
    cout << templated_func(10);
}

// -- file2.cpp --
using namespace std;

extern template int templated_func(const int&);

void g()
{
    cout << templated_func(1234);
}
```

extern Templates

```
extern template vector<int>;

extern template vector<int>::size_type vector<int>::size() const;
```

Expressions in Template Deduction Contexts

```
template<int N>
struct A
{
    static int size() { return N; }
};

int f(int);
double f(double);

template <typename T>
A<sizeof(f((T)0))> calc_size(T)
{
    return A<sizeof(f((T)0))>();
}
```

Expressions in Template Deduction Contexts

- Processing external entities
- Implementation limits
- Access violations

Class Features Overview

- In-class initializers for non-static data members
- Delegating constructors
- Inheriting constructors
- Default methods
- Deleted methods
- override and final specifiers
- Extended friend declarations
- Nested class access rights

In-class Initializers for Non-static Data Members

```
class JetPlane
{
public:
    string _model = "Unknown";
    vector<Engine> _engines {Engine(), Engine()};
};
```

```
class JetPlane
{
    vector<Engine> _engines;
    string _manufacturer;
    string _model;
public:
    JetPlane() :
        _engines(2), _manufacturer("Unknown"), _model("Unknown")
    {}

    JetPlane(const string& manufacturer) :
        _engines(2), _manufacturer(manufacturer), _model("Unknown")
    {}
};
```

In-class Initializers for Non-static Data Members

```
class JetPlane
{
    vector<Engine> _engines {Engine(), Engine()};
    string _manufacturer = "Unknown";
    string _model = "Unknown";
public:
    JetPlane()
    {}

    JetPlane(const string& manufacturer) : _manufacturer(manufacturer)
    {}
};
```

In-class Initializers for Non-static Data Members

```
class JetPlane
{
public:
    string _manufacturer = "Unknown";
    string _model = "Unknown";
    vector<Engine> _engines {get_engine_count(_manufacturer, _model)};

    static size_t get_engine_count(const string& manufacturer,
        const string& model);
};
```

In-class Initializers for Non-static Data Members

```
struct Counter
{
    int _count = 1;
};

Counter c = {10};



class JetPlane
{
public:
    vector<Engine> _engines {2};

    JetPlane() : _engines(4)
    {}
};
```

Inheriting Constructors

```
class Plane
{
    vector<Engine> _engines;
    string _manufacturer;
    string _model;
public:
    Plane(const string& manufacturer);
    Plane(const PlaneID& tail_number);
};

class JetPlane : public Plane
{
public:
    // boring
    JetPlane(const string& manufacturer) : Plane(manufacturer)
    {}

    // boring
    JetPlane(const PlaneID& tail_number) : Plane(tail_number)
    {}
};
```

Inheriting Constructors

```
class JetPlane : public Plane
{
    using Plane::Plane;
};

JetPlane plane("Boeing");   // OK


class PropPlane : public Plane
{
public:
    using Plane::Plane;

    // overrides Plane constructor with the same parameters
    PropPlane(const string& manufacturer) : Plane(manufacturer)
    {
        cout << "In PropPlane()" << endl;
    }
};

PropPlane prop_plane("ATR");
```

Inheriting Constructors

```
class Plane
{
    string _manufacturer;
public:
    Plane(const string& manufacturer) : _manufacturer(manufacturer)
    {}
};

class Boat
{
    string _boat_manufacturer;
public:
    Boat(const string& manufacturer) : _boat_manufacturer(manufacturer)
    {}
};

class FloatPlane : public Plane, public Boat
{
    using Plane::Plane;
    using Boat::Boat;

    FloatPlane(const string& manufacturer) : Plane(manufacturer), Boat("n/a")
    {}
};
```

Inheriting Constructors

```
class PropPlane : public Plane
{
    size_t _prop_count;
public:
    using Plane::Plane;
};

// oops, _prop_count is not initialized
PropPlane prop_plane("ATR");
```

Delegating Constructors

```
class JetPlane
{
    vector<Engine> _engines;
    string _manufacturer;
    string _model;
public:
    JetPlane() : _engines(2), _manufacturer("Unknown"), _model("Unknown")
    {
        configure_engines();
    }

    JetPlane(const string& manufacturer, const string& model) :
        _engines(Lookup::engine_count(manufacturer, model)),
        _manufacturer(manufacturer), _model(model)
    {
        configure_engines();
        assign_tail_number();
    }

    // ...
};
```

Delegating Constructors

```
class JetPlane
{
    vector<Engine> _engines;
    string _manufacturer;
    string _model;
public:
    JetPlane() : JetPlane(2, "Unknown", "Unknown")
    {}

    JetPlane(const string& manufacturer, const string& model) :
        JetPlane(Lookup::engine_count(manufacturer, model), manufacturer, model)
    {
        assign_tail_number();
    }
private:
    JetPlane(size_t engine_count, const string& manufacturer,
        const string& model) :
        _engines(engine_count), _manufacturer(manufacturer), _model(model)
    {
        configure_engines();
    }
    // ...
};
```

Default Methods

```
class JetPlane
{
public:
    JetPlane() = default;
    JetPlane(const JetPlane& other);
    JetPlane(JetPlane&&) = default;
};
```

```
class JetPlane
{
public:
    JetPlane() = default;

    virtual ~JetPlane() = default;

protected:
    JetPlane(const JetPlane& other) = default;
    JetPlane& operator=(const JetPlane&) = default;
};
```

Deleted Methods

```
class JetPlane
{
public:
    JetPlane() = default;
    JetPlane(const JetPlane&) = delete;
    JetPlane& operator=(const JetPlane&) = delete;
    JetPlane(JetPlane&&) = default;
    JetPlane& operator=(JetPlane&&) = default;
};
```

- Disable some instantiations of a template
- Disable unwanted conversion
- Disable heap allocation

Deleted Methods

```
template<typename T>
void serialize(const T& obj)
{
    cout << obj.to_string();
};

// PasswordStore not allowed to be serialized
void serialize(const PasswordStore&) = delete;


class Altimeter
{
public:
    Altimeter(double) {}
    Altimeter(int) = delete;
};



class StackOnly
{
public:
    void* operator new(size_t) = delete;
};
```

override and final

```
int override = 5;        // OK
int final = 10;          // OK


struct Base
{
    virtual void f(int)
    {}
};

struct Derived : public Base
{
    virtual void f(int) override        // OK
    {}

    virtual void f(double) override     // error
    {}
};
```

override and final

```
struct Base final
{};

struct Derived : public Base    // compile error, can't inherit from
{};                             // final class
```

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

struct Base : public Interface
{
    virtual void f() final
    {}
};

struct Derived : public Base
{
    virtual void f()            // compile error, can't override
    {}                          // a final method
};
```

Extended Friend Declarations

```
class A;
class B;

class Friend
{
    friend class A;   // old declarations are still OK
    friend B;         // you can also do this now
};


class Amigo
{

    friend class D;   // OK: declares new class D
    friend D;         // error: undeclared class D
};


class B;
typedef B B2;

class Amigo
{
    friend B2;        // OK
};
```

Extended Friend Declarations

```
template <typename T, typename U>
class Ami
{
    friend T;             // OK

    friend class U;       // still an error, can't use an elaborate specifier
                          // in a template
};


Ami<string, string> rc;    // OK
Ami<char, string> f;       // OK, "friend char" has no effect in the template
```

Nested Class Access Rights

```
class JetPlane
{
    // ...
private:
    int _flap_angle;

    class GPSNavigator {};

    class Autopilot
    {
        GPSNavigator _gps_navigator;        // OK, JetPlane::Autopilot can access
                                            // JetPlane::GPSNavigator
        void adjust_flaps(JetPlane& plane, int flap_angle)
        {
            plane._flap_angle = flap_angle; // OK, JetPlane::Autopilot can
                                            // access JetPlane::_flap_angle
        }
    };
};
```

The Dream of Uniform Initialization

```
class Point
{
public:
    int _x, _y;
    Point(int x, int y) : _x(x), _y(y)
    {}
};

Point p = {10, 20};
Point p(10, 20);    // have to use this instead
```

```
int values[] = {1, 2, 3}; // OK

int* p_values = new int[3] {1, 2, 3}; // not going to happen
```

```
class Hexagon
{
    int _points[6];

    Hexagon()  // no way to initialize _points in initialization list
    {}
};
```

The Dream of Uniform Initialization

```
int x(10);

int y = 20;

int values[] = {1, 2, 3};
```

Embrace the Braces

```
int x {5};

int* p_values = new int[3] {1, 2, 3};


class Point
{
public:
    int _x, _y;
    Point(int x, int y) : _x(x), _y(y)
    {}
};

Point p1 {10, 20};
Point p = {10, 20};


class Hexagon
{
    int _points[6];

    Hexagon() : _points {1, 2, 3, 4, 5, 6}
    {}
};
```

Embrace the Braces

```
vector<int> v {1, 2, 3, 4};


vector<int> extract_core_points(const vector<int>& v)
{
    return {v.front(), v[v.size() / 2], v.back()};
}

vector<int> core_points = extract_core_points({1, 2, 3, 4, 5});
```

initializer_list

{...}  initializer_list

initializer_list

```
#include <initializer_list>


Polygon(initializer_list<int> point_indexes)
{
    if (point_indexes.size() < 3)
        throw Error("Polygons require 3 or more points");

    for_each(point_indexes.begin(), point_indexes.end(),
        [=] (int index) { _points.push_back(Lookup::point(index)); });
}


Polygon(initializer_list<int> point_indexes)
{
    for_each(begin(point_indexes), end(point_indexes),
        [=] (int index) { _points.push_back(Lookup::point(index)); });
}
```

initializer_list

```
const int* p = point_indexes.begin();
cout << p[1] << endl;
```

```
vector<int> core_points;
core_points.insert(core_points.end(), {7, 9, 11});
```

Narrowing Conversions

```
int x[] =  {1, 2.5, 3};
```

Distortions of the Uniformity Continuum

```
vector<int> v1(10);

vector<int> v2{10};
```

Want a Move-only Type in Your vector?

```
vector<unique_ptr<int>> pointers {unique_ptr<int>(new int(1))};  // error

pointers.push_back(unique_ptr<int>(new int(1)));                 // OK
```

auto + {}

```
int x = 5;

auto x {5};
```

<> + {} = ?

```
template<typename T>
void f(T);

f({1});     // error
f({1,2});   // error
```

```
template<typename T>
void f(const vector<T>&);

f({1,2,3});                  //error
f({"Template","Trouble"});   //error
```

```
f(vector<int>{1, 2, 3});
f<int>({1, 2, 3});
```

Surprising Consequences of Narrowing

```
int16_t w {0};

int16_t y = {w + 1};     // error
```

```
unsigned int x {true ? 1 : 2}; // OK

bool flag {true};
unsigned int y {flag ? 1 : 2}; // error
```

What's the Verdict?

Summary

- Template features
- Class features
- Uniform initialization and initializer_list