Function default arguments

Slingshot or Shotgun?

https://github.com/michaelbprice/default-arguments/tree/cppcon2017

The Basics

Expressions that are evaluated when there are fewer provided arguments to a function call than the number of parameters specified in the function definition.

Terminology

Function default argument or default function argument

"default argument" appears 140 times in the latest C++ working draft paper[1].

Most relevant section in the standard is [dcl.fct.default] (11.3.6 Default arguments)[1]

To mirror "default template argument", we'll use "default function argument" (DFA)

[1] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4687.pdf

Simple Example

```cpp
#include <cassert>
#include <string>
using std::string

string fn (string s = "foo")
{
   return s;
}

int main ()
{
    assert(fn("foobar") == "foobar");
    assert(fn() == "foo");
    return 0;
}
```

examples/slide7.cpp

Default function arguments can not...

Default function arguments can not...

- appear in operator functions (except operator())
- appear in a position where there is already a visible DFA
- appear in a position where all parameters to the right do not have effective DFAs
- appear in out-of-class definitions of member functions of class templates
- appear in a friend declaration, unless that declaration is the only one in the translation unit and is an in-class friend function definition
- appear in declarations of
  - pointers or references to functions
  - type alias declarations
- appear in requires expressions (concepts)
- appear in explicitly defaulted member functions
- appear in user-defined literal declarations/definitions
- be provided for the first parameter of special member functions
- be provided for the first parameter of an initializer_list constructor
- be provided for the size_t parameter of allocation functions (i.e. new)
- be provided for a parameter pack
- be used to deduce a template type-parameter
- differ for an inline function defined in multiple translation units

Default function argument expressions can not contain...

Default argument expressions can not contain...

- a lambda that captures (by-value or by-reference and implicit or explicit does not matter)
- a function-local variable unless in an unevaluated context
- the this keyword
- non-static class members (with few exceptions)
- previously declared parameter names unless in an unevaluated context (but they are in scope!)

Advice: Do not use default function arguments, except in the simplest cases.

Caveat: Except maybe for the really cool uses… Maybe.

More on that later (hopefully)

An alternative to DFAs

```cpp
auto defaults (string first = "foo", string second = "bar")
{ return first + second; }

auto delegate (string first, string second)
{ return first + second; }

auto delegate (string first)
{ return delegate(first, "bar"); }

auto delegate ()
{ return delegate("foo"); }

int main ()
{
    assert(defaults() == "foobar");
    assert(defaults("baz") == "bazbar");

    assert(delegate() == "foobar");
    assert(delegate("baz") == "bazbar");
}
```

examples/slide15.cpp

But overloading is not a perfect, drop-in replacement for DFAs

```cpp
size_t index = 0;
vector<string> v = { "foo", "bar" };

string next () { return v[index++]; }

auto defaults (string first = next(), string second = next())
{ return first + second; }

auto delegate (string first, string second)
{ return first + second; }

auto delegate (string first) { return delegate(first, next()); }
auto delegate ()             { return delegate(next()); }

int main ()
{
    index = 0; assert(defaults() == "barfoo"); // WHAT!!!
    index = 0; assert(delegate() == "foobar");
    index = 0;
    assert(delegate(next(), next()) == "barfoo"); // OOPS!
}
```

examples/slide17.cpp

Mixing DFAs and overloading can cause ambiguous lookup

```
string fn (string s, int n = 42)
{
    return (n == 42) ? "foo" : s;
}

string fn (string s, bool b = false)
{
    return b ? s : "bar";
}

int main ()
{
    assert(fn("baz", 42)   == "foo");
    assert(fn("baz", true) == "baz");

    // Fails to compile
    //assert(fn("foobar")    == "???");
}
```

examples/slide19.cpp

Scope, Lookup, and Multiple Declarations

Names in a DFA are bound at declaration, but evaluated at use

```cpp
string b = "foo";

namespace N {
    string fn (string s = b) { return s; }
    string b = "bar";
}

int main ()
{
    assert(N::fn()     == "foo");
    assert(N::fn(b)    == "foo");
    assert(N::fn(N::b) == "bar");
    b = "foobarbaz";
    assert(N::fn()     == "foobarbaz");
}
```

DFAs can be provided across multiple declarations of the same function

```cpp
auto fn (string s, bool b = true)
{
    return (b) ? s : "";
}

auto fn (string s = "foo", bool b);

int main ()
{
    assert(fn() == "foo");
    assert(fn("bar") == "bar");
    assert(fn("baz", false) == "");
}
```

examples/slide24.cpp

Restrictions on DFAs across multiple declarations

- For each parameter for function **F**, there may be only a single declaration that provides a DFA.
- A parameter **P** that has a DFA in a declaration for function **F**, is allowed only if there are *visible* DFAs for all following parameters for function **F**.

- For function **F**, called within scope **S**, the *effective* DFAs for **F** are the union of all *visible* DFAs at the call-site.

The first declaration in a scope hides any previously provided DFAs

Restrictions on DFAs across multiple declarations (revisited)

- For each parameter for function **F**, there may be only a single declaration that provides a DFA.
- A parameter **P** that has a DFA in a declaration for function **F**, is allowed only if there are *visible* DFAs for all following parameters for function **F**.
- Within scope **S**, the first declaration for function **F** hides any previously *visible* DFAs for function **F** within scope **S**.
- For function **F**, called within scope **S**, the *effective* DFAs for **F** are the union of all *visible* DFAs at the call-site.

```cpp
string fn (string s = "foo", bool b = true)
{
    return (b) ? s : "";
}

int main ()
{
    assert(fn() == "foo");

  {
    string fn (string, bool); // Hides previous DFAs!!!
    //assert(fn()      == "foo"); // No longer compiles!
    //assert(fn("bar") == "bar"); // No longer compiles!
  }

  {
    string fn (string, bool = false);  // Hides previous DFAs!!!
    string fn (string = "baz", bool);
    // Both parameters now have DFAs.
    assert(fn() == "");
  }
}
```

using declarations can cause surprises

(Core issues 1551 and 1907)

```cpp
namespace N {
    auto fn (string a, string b = "bar") { return a + b; }
}

using N::fn;
void in_the_middle () { assert(fn("foo") == "foobar"); }

struct C { };
namespace N {
    // using-decl makes this next, new DFA visible outside N...
    auto fn (string = "foo", string);
    auto fn (C c) { return "C"; } // but not these overloads
    auto fn () { return  ""; }
}

int main ()
{
    in_the_middle();
    assert(fn() == "foobar"); // Not ambiguous!
    //assert(fn(C()) == "C");   // Does not compile!
    assert(N::fn(C()) == "C");
}
```

examples/slide30.cpp

DFAs on base member functions are visible...

```cpp
struct Base
{
    auto fn (string s = "foo")
    { return s; }
};

struct Derived : Base { };

int main ()
{
    Derived d;
    assert(d.fn() == "foo");
    assert(d.Base::fn() == "foo");
}
```

examples/slide32.cpp

… unless you hide them with your own member function declarations...

```
struct Base
{
    auto fn (string s = "foo") { return s; }
};

struct Derived : Base
{
    auto fn (bool) { return "char"; } // Hides Base::fn!
};

int main ()
{
    Base b; assert(b.fn() == "foo");

    Derived d;
  //assert(d.fn()       == "foo"); // Does not compile!
    assert(d.Base::fn() == "foo");
    assert(d.fn("foo")  == "char");
    // Slightly unnerving!
}
```

examples/slide34.cpp

… but you can unhide them with using-declarations in the class definition

```
struct Base { auto fn (string s = "foo") { return s; } };

struct D_one : Base {
    void fn (char);
    using Base::fn; // Bring Base names back into overload set!
};

struct D_two : Base {
    using Base::fn;
    auto fn (string s = "bar")
    { return s + "!!!"; } // Hides Base::fn!
};

int main ()
{
    Base b; assert(b.fn() == "foo");

    D_one d_one; assert(d_one.fn() == "foo");

    D_two d_two; assert(d_two.fn()       == "bar!!!");
                 assert(d_two.Base::fn() == "foo");
}
```

examples/slide36.cpp

DFAs on overridden virtual member functions

Virtual methods and DFA do not play nicely with each other

```
struct Base
{
    virtual string fn (string s = "foo") { return s; }
};

struct Derived : Base
{
    virtual string fn (string s) override
    { return s + "!!!"; }
};

int main ()
{
    Base b;
    assert(b.fn() == "foo");

    Derived d;
    //assert(d.fn() == "foo!!!"); // Does not compile!
}
```

examples/slide39.cpp

Well, that’s annoying. What can we do about it?

```
struct Base
{
    auto fn (string s = "foo") { return do_fn(s); }
 private:
    virtual string do_fn (string s) { return s; }
};

struct Derived : Base
{
 private:
    virtual string do_fn (string s) override
    { return s + "!!!"; }
};

int main ()
{
    Base b; assert(b.fn() == "foo");
    Derived d; assert(d.fn() == "foo!!!");
}
```

examples/slide41.cpp

But what if we provide a DFA for both virtual functions?

```
struct Base
{
    virtual string fn (string s = "foo") { return s; }
};

struct Derived : Base
{
    virtual string fn (string s = "bar") override
    { return s + "!!!"; }
};

int main ()
{
    Base b; assert(b.fn() == "foo");
    Derived d; assert(d.fn() == "bar!!!");
    Base & b_ref_to_d = d;
    assert(b_ref_to_d.fn() == "foo!!!");
}
```

Looking at either function definition alone, you might not expect you could possibly get a result of "foo!!!"

So, what can we do to avoid shooting ourselves in the foot?

```
struct Base
{
    auto fn ()      { return do_fn(get_fn_default()); }
    auto fn (string x) { return do_fn(x); }
 private:
    virtual string do_fn (string s)  { return s; }
    virtual string get_fn_default () { return "foo"; }
};

struct Derived : Base
{
 private:
    virtual string do_fn (string s) override  { return s + "!!!"; }
    virtual string get_fn_default () override { return "bar"; }
};

int main ()
{
    Base b; assert(b.fn() == "foo");
    Derived d; assert(d.fn() == "bar!!!");
    Base & b_ref_to_d = d; assert(b_ref_to_d.fn() == "bar!!!");
}
```

examples/slide46.cpp

But is that really any better?

Here Come the Templates

Default argument instantiation

See paragraph 11 of [temp.inst] (17.8.1 Template instantiation)

For a function template (and presumably a member function of a class template), DFAs are not always *completely* parsed until the template has been called in a way that requires the DFA.

```cpp
template <typename T>
auto fn (int n = get(T()))
{ return n; }

class A { };

auto get (A) { return 42; }

int main ()
{
    assert(fn<A>() == 42);
}
```

examples/slide51.cpp

But what if the parameter and the DFA are not dependent upon a template argument?

```cpp
template <typename T>
auto fn (int n = bool{42}) // MSVC = Nope!
{                          // Clang = Nope!
    return n;              // GCC = Why not!
}

int main ()
{
    assert(fn<int>(1) == 1); // GCC = Whynot!

    //assert(fn<int>() == 42); // GCC = Nope!
}
```

examples/slide53.cpp

What about a dependent default argument expression, that is invalid when types are substituted, but never used?

```cpp
template <typename T>
auto fn (int n = T{42}) // MSVC = Sure!
{                       // Clang = Sure!
    return n;           // GCC = Sure!
}

int main ()
{
    assert(fn<bool>(1) == 1); // MSVC = Why not!
                              // Clang = Why not!
                              // GCC = Why not!

    //assert(fn<bool>() == 42); // MSVC = Nope!
                                // Clang = Nope!
                                // GCC = Nope!
}
```

examples/slide55.cpp

What can be done about this mess?

Coding guidelines

Code Review

Static analysis tools

Unit testing

Questions?

But what about...

std::experimental::source_location

```cpp
#include <experimental/source_location>
#include <iostream>

using namespace std;
using loc = experimental::source_location;

void my_assert(bool test, const char* reason,
               loc location = loc::current())
{
    if (!test)
    {
        cout << "Assertion failed: " << location.file_name << ":"
             << location.line << ":" << location.column << ": "
             << "in function " << location.function_name << ": "
             << reason << std::endl;
    }
}

int main ()
{
    // will print out with correct filename,
    // line and column numbers, and function name.
    my_assert(false, "On purpose");
}
```

examples/slide61.cpp

Run-time and compile-time code paths for constexpr functions

```
enum class tag { compiletime, runtime };
tag runtime () { return tag::runtime; }

int identity (int n) { return n; }

constexpr int fn (int n, tag t = runtime())
{
    return (t == tag::runtime) ? identity(n)
                               : n * n;
}

int main ()
{
  //static_assert(fn(4) == 16, ""); // Does not compile!
    assert(fn(4) == 4);  // Selects runtime path!

    // Selects compile-time path!
    static_assert(fn(4, tag::compiletime) == 16, "");
}
```

Who was this virtual function dispatched from?

```cpp
static unordered_map<int, string> types;

template <typename T> static int register_type() // Assume REGISTER macro wrapper
{ types[T::_id] = T::name; return T::_id; }

struct Base
{
    using self_t = Base; constexpr static int _id = 1;
    constexpr static const char* name = "Base";

    virtual string fn (int id = self_t::_id) = 0;
};
REGISTER(Base);

struct Derived : Base
{
    using self_t = Derived; constexpr static int _id = 2;
    constexpr static const char* name = "Derived";

    virtual string fn (int id = self_t::_id) override { return types[id]; }
};
REGISTER(Derived);

int main ()
{
    Derived d; assert(d.fn() == "Derived");
    Base & db = d; assert(db.fn() == "Base");
}
```

examples/slide65.cpp

FIN