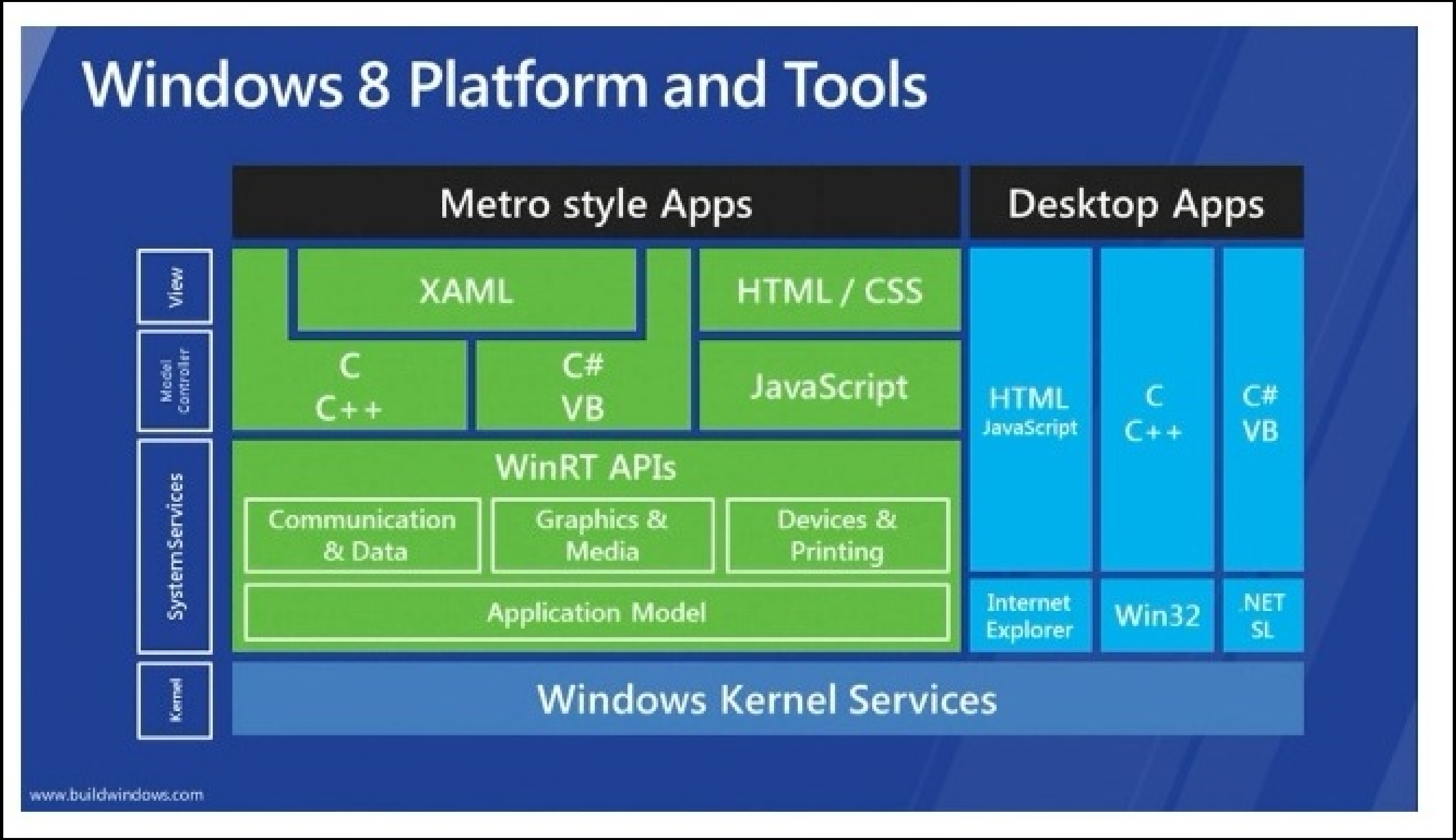
C++11 THE GOOD THE BAD THE UGLY

Frank Vasquez

“C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.”

-Bjarne Stroustrup

Welcome Back



Standards

C++98

C++03

C++0x == C++11

C++0y == C++14

Compilers

clang

gcc

Visual C++

… by and large I think it’s a bad language. It does a lot of things half well and it’s just a garbage heap of ideas that are mutually exclusive. Everybody I know, whether it’s personal or corporate, selects a subset and these subsets are different.

-Ken Thompson

Why?

classes

templates

cross-platform

threads

realtime

native

legacy code

Goodies

auto type inference

terse for loop syntax

lambdas

succinct container initialization

no more new and delete

unique data

simple async tasks

auto

```
typedef std::vector<std::pair<json::value,json::value>>
   element_vector;

json::value::element_vector m_elems;

// json::value::element_vector::iterator it;
auto it = std::begin(m_elems);
```

range-based for

```
// bad old days
for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
    sum += *it;
}

// now
for (auto& num : v) {
    sum += num;
}
```

lamda types

```
// std::function<bool (const string&)> func;

auto func = [] (const string &name) {
   return false;
};
```

lambdas

```
// Find next available UDP port
auto findUDPPort = [] (boost::asio::io_service &io) {
   udp::socket probe(io, udp::endpoint(udp::v4(), 0));
   udp::endpoint dataEndPoint = probe.local_endpoint();
   return dataEndPoint.port();
};
```

lambda as predicate

```
std::wstring month(3, L'\0');
// scanf month
std::wstring names[12] = {L"Jan", L"Feb", … , L"Dec"};
auto loc = std::find_if(std::begin(names), std::end(names),
    [&month] (const std::wstring& m) { return m == month; });

if (loc != std::end(names) {
    sysTime.wMonth = (short) ((loc - names) + 1);
}
```

lamda-friendly stl algorithms

all_of
any_of
none_of
for_each
find_if
find_if_not
count_if

copy_if
replace_if
replace_copy_if
remove_if
remove_copy_if

variable capture

```
bool _open_fsb_str(_filestream_callback *callback,
   const char *filename, std::ios_base::openmode mode, int prot)
{
    std::string name(filename);
    pplx::create_task([=] () -> void
    {
        int cmode = get_open_flags(mode);
        if (cmode==O_RDWR)  { cmode |= O_CREAT; }
        int f = open(name.c_str(), cmode, 0600);
        _finish_create(f, callback, mode, prot);
    });
    return true;
}
```

capture specification

[] Capture nothing

[&] Capture any referenced variable by reference

[=] Capture any referenced variable by making a copy

[&foo] Capture variable foo by reference

[bar] Capture just bar by making a copy

[this] Capture the this pointer of the enclosing class

initializer lists

```
/*
std::vector<string> v;
v.push_back(“rock”);
v.push_back(“paper”);
v.push_back(“scissors”);
*/
std::vector<string> v = { “rock”, “paper”, “scissors” };
```

shared_ptr

```
// shared ownership
std::shared_ptr<Quadruped> _pet;

_pet = std::make_shared<Quadruped>(“Scotty”);

// no more new or delete
// reference counting
```

weak_ptr

```
// non-owning reference to a shared_ptr
std::weak_ptr<Quadruped> wp = _pet;
if (auto sp = wp.lock()) {
   // work with unexpired shared pointer
}

// does not contribute to reference count
// breaks cycles
```

unique_ptr

```
// single owner
std::unique_ptr<Vehicle> _transport;
_transport = new Vehicle(“Ford”);

// ownership can be transferred
```

rvalue references

```
void foo(Bar && bar);

Bar snickers;   // lvalue
foo(snickers);  // compiler error

foo(Bar());  // cool
// no deep copy
// destructor only called once
```

move semantics

```
// move constructor
Buffer(Buffer&& temp)
{
   std::swap(*this, temp);
}

string lvalue = “movable”;
v.push_back(std::move(lvalue));
```

… a folk definition of insanity is to do the same thing over and over again and to expect the results to be different. By this definition, we in fact require that programmers of multithreaded systems be insane. Were they sane, they could not understand their programs.

-Edward Lee

Professor, EECS Department, UC Berkeley

async

```
vector<future<string>> futures;
vector<string> names = { "Futura", Dondi", "Chaka" };

for (auto& name : names) {
   futures.push_back(async([&name] { return flip(move(name)); }));
}

for (auto& ftr : futures) {
   cout << ftr.get() << endl;
}
```

async

```
vector<future<string>> futures;
vector<string> names = { "Futura", Dondi", "Chaka" };

for (auto& name : names) {
   // launch::deferred vs launch::async
   futures.push_back(async([&name] { return flip(move(name)); }));
}

for (auto& ftr : futures) {
   cout << ftr.get() << endl;
}
```

unique data

```
string flip(string && name)
{
    cout << this_thread::get_id() << endl;
    reverse(begin(name), end(name));
    return name;
}

// name ownership transferred to flip tasks
// name[s] empty in main after moving
// avoids races
```

Projects

Folly

http://github.com/facebook/folly

Casablanca

http://casablanca.codeplex.com

Cinder

http://libcinder.org

Books

C++ Concurrency in Action

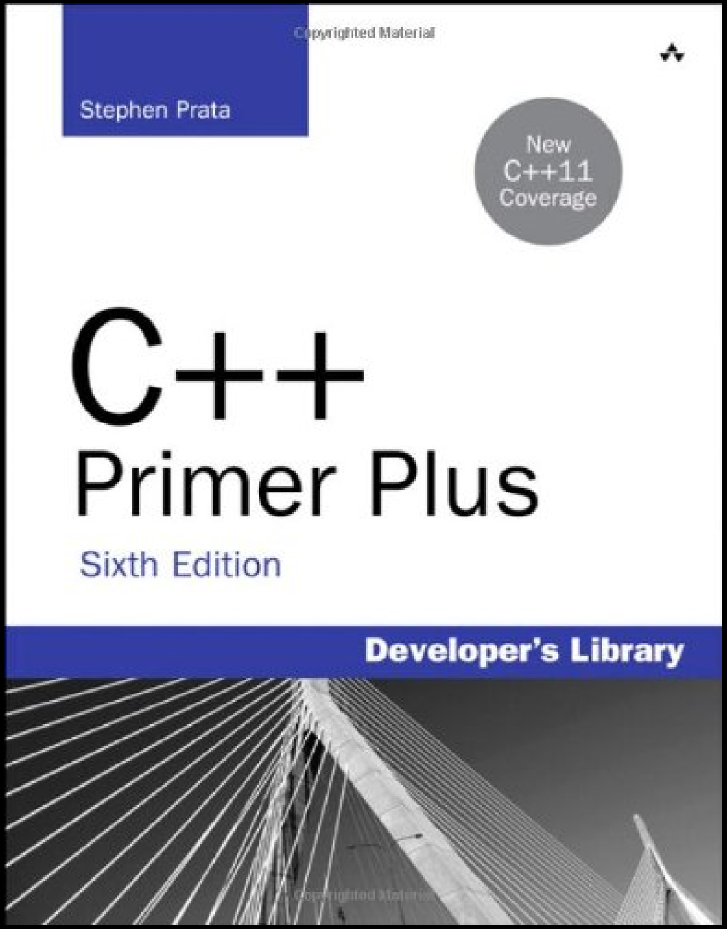
by Anthony Williams

Effective C++ series

by Scott Meyers

Exceptional C++ series

by Herb Sutter



Credits

Elements of Modern C++ Style

by Herb Sutter

http://herbsutter.com/elements-of-modern-c-style

C++11 Concurrency

9 part video series by Bartosz Milewski

http://www.youtube.com/watch?v=80ifzK3b8QQ&list