

Introduction to C++11 (by example)

This presentation gives an introduction to some of the new stuff in the C++11 standard. Some previous experience with C++ is useful, but we use only “hello, world” examples to demonstrate the new concepts in C++11 one by one. No slides - everything happens in Emacs.

Olve Maudal is a software guru working for Cisco Systems in Norway. He often give talks and courses about programming techniques, embedded systems, program design, architecture and methodologies. He has worked a lot with C and C++ the last 20 years.

30 minute presentation for students at UIO
dagen@ifi, Thursday, October 25, 2012



Introduksjon til C++11 (by example)

Dette foredraget gir en introduksjon til den nye C++11 standarden på en forhåpentligvis underholdende måte. Noe tidligere erfaring med C++ vil være nyttig, men vi vil starte med “hello, world” og deretter demonstrere de viktigste konseptene i C++11 en etter en. Ingen slides – alt skjer live i Emacs.

Olve Maudal er software guru i Cisco Systems. Han holder ofte foredrag og kurs om programmering, embedded systems, programdesign, arkitektur og arbeidsmetodikk. Han har jobbet særlig mye med C og C++ de siste 20 årene.

30 minuters presentasjon for studenter ved UIO
dagen@ifi, Torsdag 25. oktober 2012



the whole session was presented live in Emacs, the following snippets are more like a script of the presentation.

Hello! (hello1.cpp)

```
#include <iostream>
#include <string>

class foo {
public:
    foo(const std::string & msg, char p) : message(msg), punctuator(p) {}
    virtual std::string str() const { return message + punctuator; }
private:
    std::string message;
    char punctuator;
};

int main()
{
    foo f("Hello", '!');
    std::cout << f.str() << std::endl;
}
```

delegating constructor

```
#include <iostream>
#include <string>

class foo {
public:
    foo() : foo("Hello", '!') {}
    foo(const std::string & msg, char p) : message(msg), punctuator(p) {}
    virtual std::string str() const { return message + punctuator; }
private:
    std::string message;
    char punctuator;
};

int main()
{
    foo f;
    std::cout << f.str() << std::endl;
}
```

new smart pointers (unique_ptr / shared_ptr)

```
#include <iostream>
#include <string>
#include <memory>

class foo {
public:
    foo() : foo("Hello", '!') {}
    foo(const std::string & msg, char p) : message(msg), punctuator(p) {}
    virtual std::string str() const { return message + punctuator; }
private:
    std::string message;
    char punctuator;
};

class bar : public foo {
public:
    std::string str() { return "hello, world"; }
};

int main()
{
    std::unique_ptr<foo> f(new bar);
    std::cout << f->str() << std::endl;
}
```

override specifier

```
#include <iostream>
#include <string>
#include <memory>

class foo {
public:
    foo() : foo("Hello", '!') {}
    foo(const std::string & msg, char p) : message(msg), punctuator(p) {}
    virtual std::string str() const { return message + punctuator; }
private:
    std::string message;
    char punctuator;
};

class bar : public foo {
public:
    std::string str() const override { return "hello, world"; }
};

int main()
{
    std::unique_ptr<foo> f(new bar);
    std::cout << f->str() << std::endl;
}
```

final specifier

```
#include <iostream>
#include <string>
#include <memory>

class foo {
public:
    foo() : foo("Hello", '!') {}
    foo(const std::string & msg, char p) : message(msg), punctuator(p) {}
    virtual std::string str() const { return message + punctuator; }
private:
    std::string message;
    char punctuator;
};

class bar : public foo {
public:
    std::string str() const override final { return "hello, world"; }
};

int main()
{
    std::unique_ptr<foo> f(new bar);
    std::cout << f->str() << std::endl;
}
```


Hello (hello2.cpp)

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<char> msg = {'H', 'e', 'l', 'l', 'o', '!'};
    for (size_t i=0; i != msg.size(); i++)
        std::cout << msg[i];
    std::cout << std::endl;
}
```

iterator

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<char> msg = {'H', 'e', 'l', 'l', 'o', '!'};
    for (std::vector<char>::iterator it = msg.begin(); it != msg.end(); ++it)
        std::cout << *it;
    std::cout << std::endl;
}
```

const iterator

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<char> msg = {'H', 'e', 'l', 'l', 'o', '!'};
    for (std::vector<char>::const_iterator it = msg.cbegin(); it != msg.cend(); ++it)
        std::cout << *it;
    std::cout << std::endl;
}
```

decltype

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<char> msg = {'H', 'e', 'l', 'l', 'o', '!'};
    for (decltype(msg.cbegin()) it = msg.cbegin(); it != msg.cend(); ++it)
        std::cout << *it;
    std::cout << std::endl;
}
```

auto

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<char> msg = {'H','e','l','l','o','!'};
    for (auto it = msg.cbegin(); it != msg.cend(); ++it)
        std::cout << *it;
    std::cout << std::endl;
}
```

uniformed initialization

```
#include <iostream>
#include <vector>

int main()
{
    auto msg = std::vector<char>{'H', 'e', 'l', 'l', 'o', '!'};
    for (auto it = msg.cbegin(); it != msg.cend(); ++it)
        std::cout << *it;
    std::cout << std::endl;
}
```

create a greeting function

```
#include <iostream>
#include <vector>

std::vector<char> greeting()
{
    return {'H', 'e', 'l', 'l', 'o', '!'};
}

int main()
{
    auto msg = greeting();
    for (auto it = msg.cbegin(); it != msg.cend(); ++it)
        std::cout << *it;
    std::cout << std::endl;
}
```

range based for loop

```
#include <iostream>
#include <vector>

std::vector<char> greeting()
{
    return {'H', 'e', 'l', 'l', 'o', '!'};
}

int main()
{
    auto msg = greeting();
    for (char ch : msg)
        std::cout << ch;
    std::cout << std::endl;
}
```


greeting as a temporary object

```
#include <iostream>
#include <vector>

std::vector<char> greeting()
{
    return std::vector<char>{'H', 'e', 'l', 'l', 'o', '!'};
}

int main()
{
    for (auto ch : greeting())
        std::cout << ch;
    std::cout << std::endl;
}
```

creating a print() function

```
#include <iostream>
#include <vector>

std::vector<char> greeting()
{
    return {'H', 'e', 'l', 'l', 'o', '!'};
}

void print(std::vector<char> msg)
{
    for (auto ch : msg)
        std::cout << ch;
    std::cout << std::endl;
}

int main()
{
    print(greeting());
}
```

sort_and_print()

```
#include <iostream>
#include <vector>
#include <algorithm>

std::vector<char> greeting()
{
    return {'H', 'e', 'l', 'l', 'o', '!'};
}

void sort_and_print(std::vector<char> msg)
{
    std::sort(msg.begin(), msg.end());
    for (auto ch : msg)
        std::cout << ch;
    std::cout << std::endl;
}

int main()
{
    sort_and_print(greeting());
}
```

is_sorted (with Huh?)

```
#include <iostream>
#include <vector>
#include <algorithm>

std::vector<char> greeting()
{
    return {'H', 'e', 'l', 'l', 'o', '!'};
}

void sort_and_print(std::vector<char> msg)
{
    std::sort(msg.begin(), msg.end());
    for (auto ch : msg)
        std::cout << ch;
    std::cout << std::endl;
}

int main()
{
    auto msg = greeting();
    sort_and_print(msg);
    if (!std::is_sorted(msg.begin(), msg.end()))
        std::cerr << "Huh?" << std::endl;
}
```

pass by ref

```
#include <iostream>
#include <vector>
#include <algorithm>

std::vector<char> greeting()
{
    return {'H', 'e', 'l', 'l', 'o', '!'};
}

void sort_and_print(std::vector<char> & msg)
{
    std::sort(msg.begin(), msg.end());
    for (auto ch : msg)
        std::cout << ch;
    std::cout << std::endl;
}

int main()
{
    auto msg = greeting();
    sort_and_print(msg);
    if (!std::is_sorted(msg.begin(), msg.end()))
        std::cerr << "Huh?" << std::endl;
}
```

pass temp value by ref (error)

```
#include <iostream>
#include <vector>
#include <algorithm>

std::vector<char> greeting()
{
    return {'H', 'e', 'l', 'l', 'o', '!'};
}

void sort_and_print(std::vector<char> & msg)
{
    std::sort(msg.begin(), msg.end());
    for (auto ch : msg)
        std::cout << ch;
    std::cout << std::endl;
}

int main()
{
    sort_and_print(greeting());
}
```

pass temp value by const ref (template error)

```
#include <iostream>
#include <vector>
#include <algorithm>

std::vector<char> greeting()
{
    return {'H', 'e', 'l', 'l', 'o', '!'};
}

void sort_and_print(const std::vector<char> & msg)
{
    std::sort(msg.begin(), msg.end());
    for (auto ch : msg)
        std::cout << ch;
    std::cout << std::endl;
}

int main()
{
    sort_and_print(greeting());
}
```

pass by rvalue

```
#include <iostream>
#include <vector>
#include <algorithm>

std::vector<char> greeting()
{
    return {'H', 'e', 'l', 'l', 'o', '!'};
}

void sort_and_print(std::vector<char> && msg)
{
    std::sort(msg.begin(), msg.end());
    for (auto ch : msg)
        std::cout << ch;
    std::cout << std::endl;
}

int main()
{
    sort_and_print(greeting());
}
```


template

```
#include <iostream>
#include <vector>
#include <algorithm>

std::vector<char> greeting()
{
    return {'H', 'e', 'l', 'l', 'o', '!'};
}

template<typename T>
void sort_and_print(std::vector<T> && v)
{
    std::sort(v.begin(), v.end());
    for (T e : v)
        std::cout << e;
    std::cout << std::endl;
}

int main()
{
    sort_and_print(greeting());
    sort_and_print<int>({4, 1, 3, 4});
}
```

comp_char comp_int

```
#include <iostream>
#include <vector>
#include <algorithm>

std::vector<char> greeting()
{
    return {'H','e','l','l','o','!'};
}

template<typename T, typename F>
void sort_and_print(std::vector<T> && v, F comp)
{
    std::sort(v.begin(), v.end(), comp);
    for (T e : v)
        std::cout << e;
    std::cout << std::endl;
}

bool comp_char(char a, char b)
{
    return a < b;
}

bool comp_int(int a, int b)
{
    return a > b;
}

int main()
{
    sort_and_print(greeting(), comp_char);
    sort_and_print<int>({4,1,3,4}, comp_int);
}
```

std::greater and lambda

```
#include <iostream>
#include <vector>
#include <algorithm>

std::vector<char> greeting()
{
    return {'H', 'e', 'l', 'l', 'o', '!'};
}

template<typename T, typename F>
void sort_and_print(std::vector<T> && v, F comp)
{
    std::sort(v.begin(), v.end(), comp);
    for (T e : v)
        std::cout << e;
    std::cout << std::endl;
}

int main()
{
    sort_and_print(greeting(), std::greater<char>());
    sort_and_print<int>({4,1,3,4}, [](int a, int b) { return a > b; });
}
```

lambda and auto

```
#include <iostream>
#include <vector>
#include <algorithm>

std::vector<char> greeting()
{
    return {'H', 'e', 'l', 'l', 'o', '!'};
}

template<typename T, typename F>
void sort_and_print(std::vector<T> && v, F comp)
{
    std::sort(v.begin(), v.end(), comp);
    for (T e : v)
        std::cout << e;
    std::cout << std::endl;
}

int main()
{
    sort_and_print(greeting(), std::greater<char>());
    sort_and_print<int>({4,1,3,4}, [](int a, int b) { return a > b; });
    auto the_answer = []() { return 42; };
    std::cout << the_answer() << std::endl;
}
```

capture

```
#include <iostream>
#include <vector>
#include <algorithm>

std::vector<char> greeting()
{
    return {'H', 'e', 'l', 'l', 'o', '!'};
}

template<typename T, typename F>
void sort_and_print(std::vector<T> && v, F comp)
{
    std::sort(v.begin(), v.end(), comp);
    for (T e : v)
        std::cout << e;
    std::cout << std::endl;
}

int main()
{
    sort_and_print(greeting(), std::greater<char>());
    sort_and_print<int>({4,1,3,4}, [](int a, int b) { return a > b; });
    int seed = 7;
    auto the_answer = [=]() { return seed*6; };
    std::cout << the_answer() << std::endl;
}
```

clean up (hello3.cpp)

```
#include <iostream>
#include <vector>
#include <algorithm>

std::vector<char> greeting()
{
    return {'H', 'e', 'l', 'l', 'o', '!'};
}

void print(std::vector<char> msg)
{
    for (auto ch : msg)
        std::cout << ch;
}

int main()
{
    std::vector<char> msg = greeting();
    print(msg);
}
```

future and async

```
#include <iostream>
#include <vector>
#include <future>

std::vector<char> greeting()
{
    // sleep for a while
    return {'H','e','l','l','o','!'};
}

void print(std::vector<char> msg)
{
    for (auto ch : msg)
        std::cout << ch;
    std::cout << std::endl;
}

int main()
{
    std::future<std::vector<char>> msg = std::async(greeting);
    // do something else
    print(msg.get());
}
```

sleep_for

```
#include <iostream>
#include <vector>
#include <future>

std::vector<char> greeting()
{
    // std::this_thread::sleep_for(std::chrono::seconds 2);
    return {'H', 'e', 'l', 'l', 'o', '!'};
}

void print(std::vector<char> msg)
{
    for (auto ch : msg)
        std::cout << ch;
    std::cout << std::endl;
}

int main()
{
    std::future<std::vector<char>> msg = std::async(std::launch::deferred, greeting);
    // do something else
    print(msg.get());
}
```


uniformed initialization

```
#include <iostream>
#include <string>

int main()
{
    std::string msg{"Hello!"};
    std::cout << msg << std::endl;
}
```

simple

```
#include <iostream>

int main()
{
    std::cout << "Hello!" << std::endl;
}
```

echo on command line

```
$ echo 'Hello!'
```

overview

Hello (hello1.cpp)
delegating constructor
unique_ptr
override
final
Hello (hello2.cpp)
iterator
const iterator
decltype
auto
uniformed initialization
create a greeting function
range based for loop
greeting as a temporary object
creating a print() function
sort_and_print()
is_sorted (with Huh?)

pass by ref
pass temp value by ref (error)
pass temp value by const ref (template error)
pass by rvalue
template
template
comp_char comp_int
std::greater and lambda
lambda and auto
capture
clean up (hello3.cpp)
future and async
sleep_for
uniformed initialization
simple
echo on command line

summary

delegating constructor

improved smart pointers (unique_ptr and shared_ptr)

override and final specifiers

decltype and auto

range based for loops

uniformed initialization

rvalue references and move semantics

some new library functions

multi argument templates

lambda functions

chrono

futures and promises

not shown:

regex

user defined literals

and lots of smaller stuff