Coding Without Words

Roger Orr

OR/2 Limited

Some strengths and weaknesses of C++ programming without explicitly **naming** things.

– ACCU 2015 –

What's in a name?

- There are various features of C++ that allow you to avoid providing your own names
    - lambda (unnamed functions)
    - tuple (unnamed members)
    - auto (unnamed types)
- This can help with *genericity* and *brevity*
- Let's see how we get on in practice ...

What's in a name?

- One of the motivations for this talk was, somewhat simplified:

```
bool A::doit()
{
    return _coll->apply([&](value b) {
        if (/* ... */) {
            // ...
            return true;
        }
        log("Failed to process " + to_string(b));
        return false;
    });
}
```

- It was easy to miss that the `return false` **didn't** actually mean the value returned from `doit`

What's in a name?

- To make it even more galling, the `apply` method was actually taking `std::function<void(value)>` so in fact the inner return value was being *completely* ignored
- My interest though was what was it about the use of the lambda that seemed to change the way the code was being **read** – in particular that the return statement exited the lambda rather than the enclosing function

What's in a name?

- We usually have a choice between a named and an unnamed way of writing the code
- What are some of the issues we should think about when making these choices?
- For obvious reasons most of the examples are short; however most of the programs we actually work on are not: there are some larger scale issues at work too as well as the more local ones. I hope to mention both.

Example 'straw man' program

- Let's start with a simple program and implement it in a number of ways...

```
#include <fstream>
#include <iostream>

#include "readnames.h"
#include "sortnames.h"
#include "printnames.h"

int main()
{
    std::ifstream ifs("names.txt");
    Names names;

    readnames(ifs, names);
    sortnames(names);
    printnames(std::cout, names);
}
```

Headers – old school C++

Possible data types:

```
struct Name
{
    std::string first;
    std::string last;
};

typedef std::vector<Name> Names; // don't worry about the precise collection type!
```

The corresponding function declarations:

```
void readnames(std::istream & is, Names & names);

void sortnames(Names & names);

void printnames(std::ostream & os, Names const & names);
```

(Note that the declarations don't explicitly refer to `Name`, `first` or `last`)

readnames – old school C++

```
void readnames(std::istream & is, Names & names)
{
    Name next; // or Names::value_type next;
    while (is >> next.first >> next.last)
    {
        names.push_back(next);
    }
}
```

We might prefer to factor out a helper function:

```
std::istream & operator>>(std::istream & is, Names::value_type & name)
{
    return is >> name.first >> name.last;
}

void readnames(std::istream & is, Names & names)
{
    Names::value_type next;
    while (is >> next)
    {
        names.push_back(next);
    }
}
```

sortnames – old school C++

- Using a functor

```
struct first_last
{
    bool operator()(Name const & lhs, Name const & rhs)
    {
        if (lhs.first < rhs.first)
            return true;
        else if (lhs.first == rhs.first)
            return lhs.last < rhs.last;
        return false;
    }
};

void sortnames(Names & names)
{
    std::sort(names.begin(), names.end(), first_last());
}
```

sortnames – old school C++

- Or use operator< to remove another name

```
bool operator<(Name const & lhs, Name const & rhs)
{
    if (lhs.first < rhs.first)
            return true;
    else if (lhs.first == rhs.first)
        return lhs.last < rhs.last;
    return false;
}

void sortnames(Names & names)
{
    std::sort(names.begin(), names.end());
}
```

printnames – old school C++

- We could use `for_each`:

```
namespace
{
    class print
    {
        std::ostream & os;
    public:
        print(std::ostream &os) : os(os) {}
        void operator()(Names::value_type const &name)
        { os << name.first << ' ' << name.last << '\n'; }
    };
}

void printnames(std::ostream & os, Names const & names)
{
    std::for_each(names.begin(), names.end(), print(os));
}
```

printnames – old school C++

- Or use another operator: `operator<<`

```
std::ostream & operator<<(std::ostream & os, Names::value_type const & name)
{
    return os << name.first << ' ' << name.last;
}

void printnames(std::ostream & os, Names const & names)
{
    std::copy(names.begin(), names.end(), std::ostream_iterator<Names::value_type>(os, "\n"));
}
```

What's good and bad so far?

- Pro:
  - The code is simple to understand
  - The meaning of the code is clear
- Con:
  - It would be hard to re-use – the *algorithms* are generic but the names are very specific
  - Lots of repetition – more work to write & read
- Can we do better?

Reducing scope: C++11

- Local classes

```
void printnames(std::ostream & os, Names const & names)
{
    class print
    {
        std::ostream & os;
    public:
        print(std::ostream &os) : os(os) {}
        void operator()(Names::value_type const &name)
        { os << name.first << ' ' << name.last << '\n'; }
    };

    std::for_each(names.begin(), names.end(), print(os));
}
```

We can now use a *local* name and it is scoped inside the method using it.
But we've still got to name it – but only in this scope.

Reducing scope: C++11

- Local classes

```
void printnames(std::ostream & os, Names const & names)
{
    class f
    {
        std::ostream & os;
    public:
        f(std::ostream &os) : os(os) {}
        void operator()(Names::value_type const &name)
        { os << name.first << ' ' << name.last << '\n'; }
    };

    std::for_each(names.begin(), names.end(), f(os));
}
```

We can now use a *local* name and it is scoped inside the method using it. Since the scope is very restricted we can use a “placeholder” name.

This proposal was first formally made in 2001 (by Anthony Williams) and adopted into C++11 in 2008 after five further papers. Phew...

Changing code: C++11

- lambda

```
void printnames(std::ostream & os, Names const & names)
{
    std::for_each(names.begin(), names.end(), [&os](Name const & name)
    { os << name.first << ' ' << name.last << '\n'; });
}
```

We don't need to name the target functoid of '`for_each`'

C++11

- This lambda is (roughly) equivalent to

```
void printnames(std::ostream & os, Names const & names)
{
    class unnamed {
        std::ostream &os;
    public:
        unnamed(std::ostream &os) : os(os) {}
        void operator()(Name const & name)
        { os << name.first << ' ' << name.last << '\n'; }
   };

    std::for_each(names.begin(), names.end(), unnamed(os));
}
```

- So, if we understand the previous example, lambda is 'easy'; it is just that the compiler
    - names it
    - writes much of the scaffolding

C++14

- Generic lambda

```
void printnames(std::ostream & os, Names const & names)
{
    std::for_each(names.begin(), names.end(), [&os](auto name)
    { os << name.first << ' ' << name.last << '\n'; });
}
```

- We can remove the use of the name of the contained type.
- Except we might need to think rather than naively just use auto...

C++14

- Generic lambda

```
void printnames(std::ostream & os, Names const & names)
{
    std::for_each(names.begin(), names.end(), [&os](auto const & name)
    { os << name.first << ' ' << name.last << '\n'; });
}
```

- The original proposals for generic lambda used the term 'polymorphic lambda', but in this case we're not actually using the lambda in a polymorphic fashion at all. In my experience this is a common use case for generic lambdas.

C++14

- The generic lambda is roughly equivalent to

```
void printnames(std::ostream & os, Names const & names)
{
    class unnamed {
        std::ostream &os;
    public:
        unnamed(std::ostream &os) : os(os) {}
        template <typename T>
        void operator()(T const & name)
        { os << name.first << ' ' << name.last << '\n'; }
   };

    std::for_each(names.begin(), names.end(), unnamed(os));
}
```

- A generic lambda changes the C++11 style member function into a member function *template*

Concepts TS

- The generic lambda is roughly equivalent to

```
void printnames(std::ostream & os, Names const & names)
{
    class unnamed {
        std::ostream &os;
    public:
        unnamed(std::ostream &os) : os(os) {}
        void operator()(auto const & name)
        { os << name.first << ' ' << name.last << '\n'; }
    };

    std::for_each(names.begin(), names.end(), unnamed(os));
}
```

- This makes the correspondence more uniform
- Voting is in progress on the TS
- A variation of this TS is likely to be in a future standard

Another way: C++11

- Range-for and auto

```
void printnames(std::ostream & os, Names const & names)
{
   for (auto name : names)
   {
      os << name.first << ' ' << name.second << '\n';
   }
}
```

- We don't need to concern ourselves with the iteration itself nor directly with the type 'Name'.
- In the case of `for` this is likely to be both shorter and simpler than using `for_each`
- There are however many other algorithms that haven't got language support and for these lambda is useful
- As with generic lambda earlier avoid naïve use of **auto**

Another way: C++11

- Range-for and auto

```
void printnames(std::ostream & os, Names const & names)
{
    for (auto const & name : names)
    {
        os << name.first << ' ' << name.second << '\n';
    }
}
```

- One problem with `auto` and range-based `for` is that you rarely want a plain `auto`
- You typically want a reference to the target object – just as we saw earlier for generic lambda

Digression: range for and auto

- Simpler syntax for range-for was proposed by Stephan T. Lavavej in Jan 2014 (N3853), but this has proved quite controversial
- He states the correct default for range for as **auto &&**
- The syntax originally proposed was

```
for (elem : range)
```

- The original proposal was rejected in Urbana 2014 after being provisionally accepted while C++14 was in ballot
- Since generic lambda - and concepts TS - have the same issue I'm personally less persuaded we need special syntax for the case of range for as it is something that people will need to be aware of

C++11

- Lambda with another algorithm

```
void sortnames(Names & names)
{
    std::sort(names.begin(), names.end(),
        [](Name const & lhs, Name const & rhs)
        {
            if (lhs.first < rhs.first)
                return true;
            else if (lhs.first == rhs.first)
                return lhs.last < rhs.last;
            return false;
        }
    );
}
```

One problem with lambdas is finding the best way to format them...

Datatypes

- The examples so far made small changes to the individual functions without attacking the basic datatype or interfaces.
- We can use a generic data type to avoid having to make one of our own.
- On the plus side this means less code to write / more code we get 'for free'

Datatypes: C++03

- Use pair to get rid of 'Name'

```
using Names = std::vector<std::pair<std::string, std::string>>;
```

We've saved defining, and naming, a `struct`.

```
void sortnames(Names & names)
{
    std::sort(names.begin(), names.end());
}
```

There – that was easy, wasn't it? We get an `operator<` for free!

But pair is restricted to only two items (the hint is in the name).
This is a solution that doesn't generalise to more than two fields*.

*Trying to use pairs of pairs is a shortcut to madness.

Datatypes: C++03

- Use `pair` to get rid of 'Name'

```
using Names = std::vector<std::pair<std::string, std::string>>;
```

We've saved defining, and naming, a struct.

```
void sortnames(Names & names)
{
    std::sort(names.begin(), names.end());
}
```

There – that was easy, wasn't it? We get an operator< for free!

But pair is restricted to only two items (the hint is in the name).
This is a solution that doesn't generalise to more than two fields*.

*Trying to use pairs of pairs is a shortcut to madness.

Unfortunately it hasn't stopped people trying it.

I resisted an example...

Datatypes: C++11

- Use `tuple` to get rid of 'Name'

```
using Names = std::vector<std::tuple<std::string, std::string>>;
```

We've again saved defining, and naming, a `struct`.

```
void sortnames(Names & names)
{
    std::sort(names.begin(), names.end());
}
```

There – as easy as pair, and more generic!
(We still get an `operator<` for free)

Datatypes: C++11

- Access to tuple is a little … painful

```
void printnames(std::ostream & os, Names const & names)
{
    for (auto const & name : names) {
        os << std::get<0>(name) << ' ' << std::get<1>(name) << '\n';
    }
}
```

- `std::get<0>()` is not, to my mind anyway, very readable
- Note: we have to qualify the call with `std::` as argument dependent lookup doesn't work well with templates
- However, you might have hoped `tuple` would have an `operator<<` already wouldn't you ... how hard can it be?

Datatypes: C++11

- Writing operator<< for tuple: YAGNI

```
std::ostream & operator<<(std::ostream & os, Names::value_type const & name)
{
    return os << std::get<0>(name) << ' ' << std::get<1>(name);
}
```

That's cheating: it works for our specific case only!
Can we do it for *any sized* tuple?

Datatypes: C++11

- A possible generic operator<<

```
template <size_t Pos, class... Args>
struct print_tuple {
    std::ostream& operator()(std::ostream& os, std::tuple<Args...> const & t) {
        return print_tuple<Pos-1, Args...>()(os, t) << ' ' << std::get<Pos>(t);
    }
};

template <class... Args>
struct print_tuple<0, Args...> {
    std::ostream& operator()(std::ostream& os, std::tuple<Args...> const & t) {
        return os << std::get<0>(t);
    }
};

template <class... Args>
std::ostream& operator<<(std::ostream& os, std::tuple<Args...> const & t) {
    return print_tuple<sizeof...(Args)-1, Args...>()(os, t);
}
```

Datatypes: C++11

- Add a bugfix for tuple<>

```
template <size_t Pos, class... Args>
struct print_tuple {
    std::ostream& operator()(std::ostream& os, std::tuple<Args...> const & t) {
        return print_tuple<Pos-1, Args...>()(os, t) << ' ' << std::get<Pos>(t);
    }
};

template <class... Args>
struct print_tuple<0, Args...> {
    std::ostream& operator()(std::ostream& os, std::tuple<Args...> const & t) {
        return os << std::get<0>(t);
    }
};

template <class... Args>
std::ostream& operator<<(std::ostream& os, std::tuple<Args...> const & t) {
    return print_tuple<sizeof...(Args)-1, Args...>()(os, t);
}

template <> std::ostream& operator<<(std::ostream& os, std::tuple<> const &) {
    return os;
}
```

Datatypes: C++11/14

- Using the generic operator<<

```
void printnames(std::ostream & os, Names const & names)
{
    std::for_each(names.begin(), names.end(),
    [&os](Names::value_type const & name) { os << name << '\n'; });
}
```

Or (C++14)

```
void printnames(std::ostream & os, Names const & names)
{
    std::for_each(names.begin(), names.end(), [&os](auto const & name) { os << name << '\n'; });
}
```

Looks good to me – let's ship it.

Datatypes: C++11

- Pitfall – 'Names' is no longer **our** type

This alternative **fails to compile**:

```
void printnames(std::ostream & os, Names const & names)
{
    std::copy(names.begin(), names.end(), std::ostream_iterator<Names::value_type>(os, "\n"));
}
```

The error is:

```
.../include/c++/bits/stream_iterator.h:198:13: error: cannot bind
 'std::ostream_iterator<std::tuple<std::basic_string<char, std::char_traits<char>,
  std::allocator<char> >, std::basic_string<char, std::char_traits<char>,
  std::allocator<char> > > >::ostream_type {aka std::basic_ostream<char>}' lvalue
to 'std::basic_ostream<char>&&'
  *_M_stream << __value;
```

So that's obviously what's wrong then....

Datatypes: C++11

- Pitfall – 'Names' is no longer **our** type

This alternative **fails to compile**:

```
void printnames(std::ostream & os, Names const & names)
{
   std::copy(names.begin(), names.end(), std::ostream_iterator<Names::value_type>(os, "\n"));
}
```

What the error message was trying to tell us.

The problem is that Names::value_type is tuple, which is in the **std** namespace, as is ostream_iterator and ostream – hence inside the template expansion there's nothing to allow the compiler to find the operator<< we have defined in the default namespace. (Do not try changing this!)

Note: What's potentially worse is that if some other source code has defined *another* operator<< for tuple we have an ODR violation: we're likely to get one of the implementations selected but with no clear pattern as to **which** one!

Datatypes: C++11

- Pitfall – 'Names' is no longer **our** type
- How can we fix this?
  - Inheritance

    ```
    struct Name : tuple<string, string>
    {...};
    ```

  - Using tuple in the **implementation** of < for our own class
  - But we need a **name** again

Datatypes: C++11

- Inheritance example
  - May need to explicitly inherit ctors (C++11)

```
struct Name : std::tuple<std::string, std::string>
{
    using std::tuple<std::string, std::string>::tuple;
};
```

- Now we can use the implementation of the standard type while keeping **this** type ours

Datatypes: C++11

- Using tuple for implementing operator<

```
struct Name
{
    std::string first;
    std::string last;
};

bool operator<(Name const &lhs, Name const &rhs) {
      return std::tie(lhs.first, lhs.last) <
             std::tie(rhs.first, rhs.last);
}
```

- Note this may be as fast as hand-written comparisons

Ease of change

- Code rarely stays the same; how does the presence or absence of a name help with code refactoring?
- If the change is in line with the genericity provided by the C++ feature it's easy
- For example, consider the changes needed if we add a **middle** name.
- (We'll not consider dealing with the fact that is **optional**!)

Ease of change - fields

- Adding a middle name – “old school”
- With our own struct we need to
  - add a new field to the struct
  - change the implementations of many of our functions to use the new field
- There may be some sort of refactoring support
- Alternatively, simply search for the existing field names and data types in the codebase (again, there may be tool support for this)

Ease of change - fields

- Adding a middle name – `tuple`
- With `tuple` we need to
    - Change the type to a tuple with three elements
    - **Generic** functions such as `operator<<`, `operator>>` and `operator<` 'just work'
    - Add processing for the extra element to non-generic code
- Hard to see how we can get refactoring support – it's not **our** type
- There's no field **name** to search for – we have to find the **relevant** usages of get<> and add another. We may also have to find the relevant usages of `get<1>` and change them to `get<2>`

Ease of change - functionality

- Suppose we want to sort our list of names by **last** name and then first name.
- With our own type we can simply change the function used in the sort function

```
bool last_first(Name const & lhs, Name const & rhs)
{
    if (lhs.last < rhs.last)
            return true;
    else if (lhs.last == rhs.last)
        return lhs.first < rhs.first;
    return false;
}

void sortnames(Names & names)
{
    std::sort(names.begin(), names.end(), last_first);
}
```

Ease of change - functionality

- Suppose we want to sort our list of names by **last** name and then first name.
- If we're using a standard system type and `operator<` we can't do this
  - This might encourage some dubious practice
  - “Let's read the name in reverse order”

Ease of change - functionality

- One reason I've seen `tuple` used is to take advantage of the implicit `operator<`
- Has anyone in the room ever seen an incorrectly implemented `operator<` ?
- However, since the implementation is fixed, I've seen cases where the element use is non-intuitive simply to get the comparison semantics right
- What other solutions are there?

Ease of change - functionality

- I've already mentioned using `std::tie`
- This lets you pick which fields are included in the comparison and which order

```
return std::tie(lhs.last, lhs.first) < std::tie(rhs.last, rhs.first);
```

- However, you still have to write some code, and this needs maintenance if you change the class, for example if we add a middle name...

Ease of change - functionality

- There is ongoing discussion about letting the **compiler** generate operator<
  - Oleg Smolsky (N3950 and more)
  - Bjarne Stroustrup (N4175 and more)
  - Agreement may yet be reached
  - Questions to resolve include:
    - Opt-in or opt-out?
    - Other operators?
    - Which fields are included?

Scoping issues

- Lambda allows you to *implicitly* 'capture' a variable.
- This adds a member to the (created) class

```
void printnames(std::ostream & os, Names const & names) {
    std::for_each(names.begin(), names.end(), [&](auto name) {
        os << name.first << ' ' << name.last << '\n';
    });
}
```

- The compiler-generated class contains a reference member variable `os` initialised from `os`
- But only if the name is **local** scope

Scoping issues

- Quick test (and short enough to be obvious)

```
#include <iostream>

int i{};

int main()
{
   static int j{};
   int k{};
   auto lambda = [=]() mutable {++i; ++j; ++k; };
   lambda();
   std::cout << i << ' ' << j << ' ' << k << std::endl;
}
```

- What do you expect as output?
- Scott Meyers EMC++ recommends not using capture defaults (`[=]`, `[&]`)

More fun with lambdas

- C++ provides some standard functoids

```
#include <functional>

void modify(std::vector<double> & vec)
{
   using namespace std::placeholders;

   std::transform(vec.begin(), vec.end(), vec.begin(),
      std::bind(std::plus<double>(), _1, 1));

   std::transform(vec.begin(), vec.end(), vec.begin(),
      std::bind(std::plus<>(), _1, 1)); // C++14
}
```

- We can do something similar with generic lambda in C++14

More fun with lambdas

- Here's one approach

```
void modify(std::vector<double> & vec)
{
   std::transform(vec.begin(), vec.end(), vec.begin(),
      [](auto x) { return x + 1; });
}
```

- Provides the function exactly where it is needed
- Not restricted to the set of functions predefined in the standard library

Lambdas on lambdas

- Here's another approach

```
auto plus = [](auto y) {
  return [y](auto x) {
    return x + y; };
};

void modify(std::vector<double> & vec)
{
   std::transform(vec.begin(), vec.end(), vec.begin(),
      plus(1));
}
```

- The outer lambda returns a lambda that uses the supplied argument
- Concise to write, no boiler-plate template class necessary
- Useful where the function is a little more complex than '+'

Lambdas on lambdas

- Or should it be:

```
auto plus = [](auto && y) {
  return [&y](auto && x) {
    return x + y; };
};
```

- The outer lambda returns a lambda that uses the supplied argument – but this may be a temporary
- Do we want a reference or value for 'y' in the inner lambda?
- And how do we do perfect forwarding?

Using std::forward

- With a regular template it's easy to forward

```
template <typename T> void f(T && t)
{
   g(std::forward<T>(t));
}
```

- Using `std::forward` means that the argument type of `g()` will now match the argument type of the template instantiation

Using std::forward

- How do we forward with a generic lambda?

```
auto f = [](auto && t)
{
   return g(std::forward<???>(t));
}
```

- Using a generic lambda we no longer have a name for the template argument to use with `std::forward`

Using std::forward

- How do we forward with a generic lambda?

```
auto f = [](auto && t)
{
    return g(std::forward<???>(t));
}
```

- We can use `decltype` on the variable

```
auto f = [](auto && t)
{
    return g(std::forward<decltype(t)>(t));
}
```

Type of x and decltype of x ...

- “The name of the song is called 'Haddocks' Eyes'!”
- "Oh, that's the name of the song, is it?" Alice said, trying to feel interested.
- "No, you don't understand," the Knight said, looking a little vexed. "That's what the name is called. The name really is, 'The Aged Aged Man.'"
- "Then I ought to have said "That's what the song is called'?" Alice corrected herself.
- "No, you oughtn't: that's quite another thing! The song is called 'Ways and Means': but that's only what it is called you know!"
- "Well, what is the song then?" said Alice, who was by this time completely bewildered.
- "I was coming to that," the Knight said. "The song really is "A-sitting on a Gate": and the tune's my own invention."
- -- Lewis Carroll, "Through the Looking Glass"

Variadic templates

- Some 'interesting' constructs are possible

```
template<class F, class ...Ts>
void for_each_arg(F f, Ts &&...args) {
  [](...){}((f(std::forward<Ts>(args)), 0)...);
}

void foo(int i) {std::cout << i << std::endl;}

int main() {
    for_each_arg(foo, 1, 2, 3);
}
```

- `foo` is invoked for each argument (in an undetermined order) - the results are discarded by the comma operator
- The list of 0s is passed to the varargs lambda … that does nothing with it
- Is this wonderful or dreadful – or both?

Variadic templates

- What does this code print?

```
void f(int, int) {}

int x(const char *p)
{
  std::cout << p;
  return 0;
}

int main()
{
   f(x("a"), x("b"));
}
```

- ab or ba ?

Variadic templates

- It would be nice if we had deterministic ordering with the function call
- The lack of determinism is confusing and also produces some subtly broken code
- There has been an initial proposal, N4228, to specify evaluation order
- a(b,c,d) would evaluate in the order a,b,c,d.
    - EWG showed strong support
    - Can change output for some compilers
    - May reduce optimisation

Variadic templates

- If you want deterministic ordering

```
template<class F, class ...Ts>
void for_each_arg(F f, Ts &&...args) {
  (void)std::initializer_list<int>{(f(std::forward<Ts>(args)), 0)... };
}

void foo(int i) {std::cout << i << std::endl;}

int main() {
    for_each_arg(foo, 1, 2, 3);
}
```

- `foo` is invoked for each argument as the `initializer_list` is constructed and hence the order of evaluation will be left-to-right

Some conclusions

- C++ provides some features that let you dispense with naming things
  - lambda (unnamed functions)
  - tuple (unnamed members)
  - auto (unnamed types)
- These can be very useful for simple cases
- Names are useful to express intent and can also help with type ownership and code maintenance
- There are also issues to consider with scoping and the lifetime of implicit fields