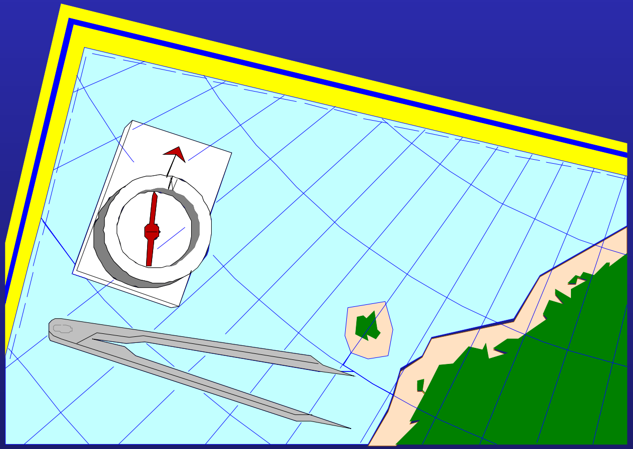
Where are we headed?

C++ style directions

Hubert Matthews
ACCU 2014
hubert@oxyware.com

April 2014



Why?

Who?

Why?

How?

Some questions

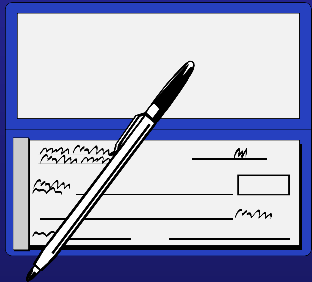
- What makes C++ different to other languages?
- What has been added in C++11 to make it even more different?
  - What we have learnt up to now and how it's all changing
- What are the implications of the new features?
  - How can we use them effectively?
  - What may we have to learn or even unlearn?
- What does C++14 add to this?
- And where possibly beyond C++14?

What makes C++ 98/03 different?

value semantics copy control conversions	→	copies + temporaries, copy c/tr, operator=, non-explicit c/tr, UDCs
destructors	→	RAII, exception safety, uniform clean up mechanism
const	→	const safety, const overloading
references	→	aliases, fixed, not null
templates type deduction specialisation	→	c/tr helpers (e.g. make_pair), generic programming (STL), metaprogramming, traits

Value semantics

- Most other languages have reference semantics (or are biased toward them)
  - Objects aren't copied; only pointers
  - Java/C#: null, NullPointerException, cloning
  - Garbage collection requires tracking which pointers are reachable
  - Wrappers around primitives; boxing/unboxing
- C++ has value semantics for every class by default
- Fits well with lexical scoping and stack allocation (original use of `auto`)



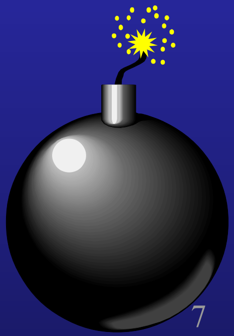
Value semantics (2)

- Conversion between value types is common
  - non-explicit c/trs: std::string(const char *)
  - user-defined conversions
- Value types using operator overloading
  - c.f. .equals() and == in Java
- STL uses “copy in, copy out” idiom
- Function parameters pass-by-value by default
- Copying and temporaries can be a performance issue (allocate, copy, deallocate)
- Exception safety issues often caused by copying

£$€

Destructors

- Probably the key feature of C++
  - uniform clean-up mechanism for normal and exceptional code (RAII)
- Implicitly called
  - you can’t forget to clean up (safe)
  - can be put in a library (again, safe)
  - don’t clutter code (code clarity)
  - consequence of value semantics (no GC)
  - c.f. Java’s try/finally or C#’s using/IDispose
- Implicitly generated – can’t not have one!

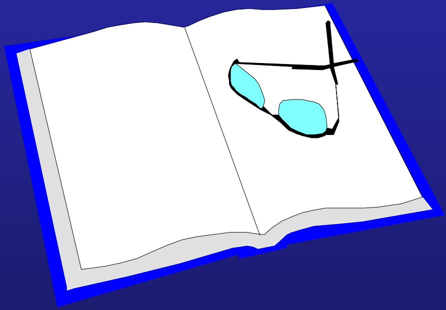


Const

- Something that C++ takes far more seriously than other languages
  - const safety is viral within a program
- Only really needed in one place in C++03
  - binding a temporary to a const reference
- Safe default pass-by-value semantics reproduced by pass-by-const-reference leads to the need for const methods
- Can overload on const

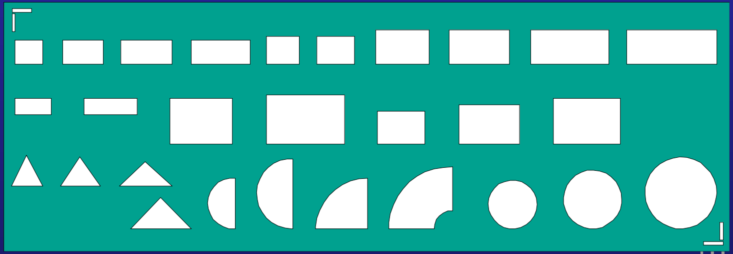
References

- Explicit aliases for objects
- Needed because of default value semantics
  - alternative is to use pointers but syntax is horrid
- Compile-time entities
  - no run-time existence (unlike pointers)
- Can't be null
  - avoids String.IfNullOrEmpty() calls
  - no NullPointerException problems



Templates

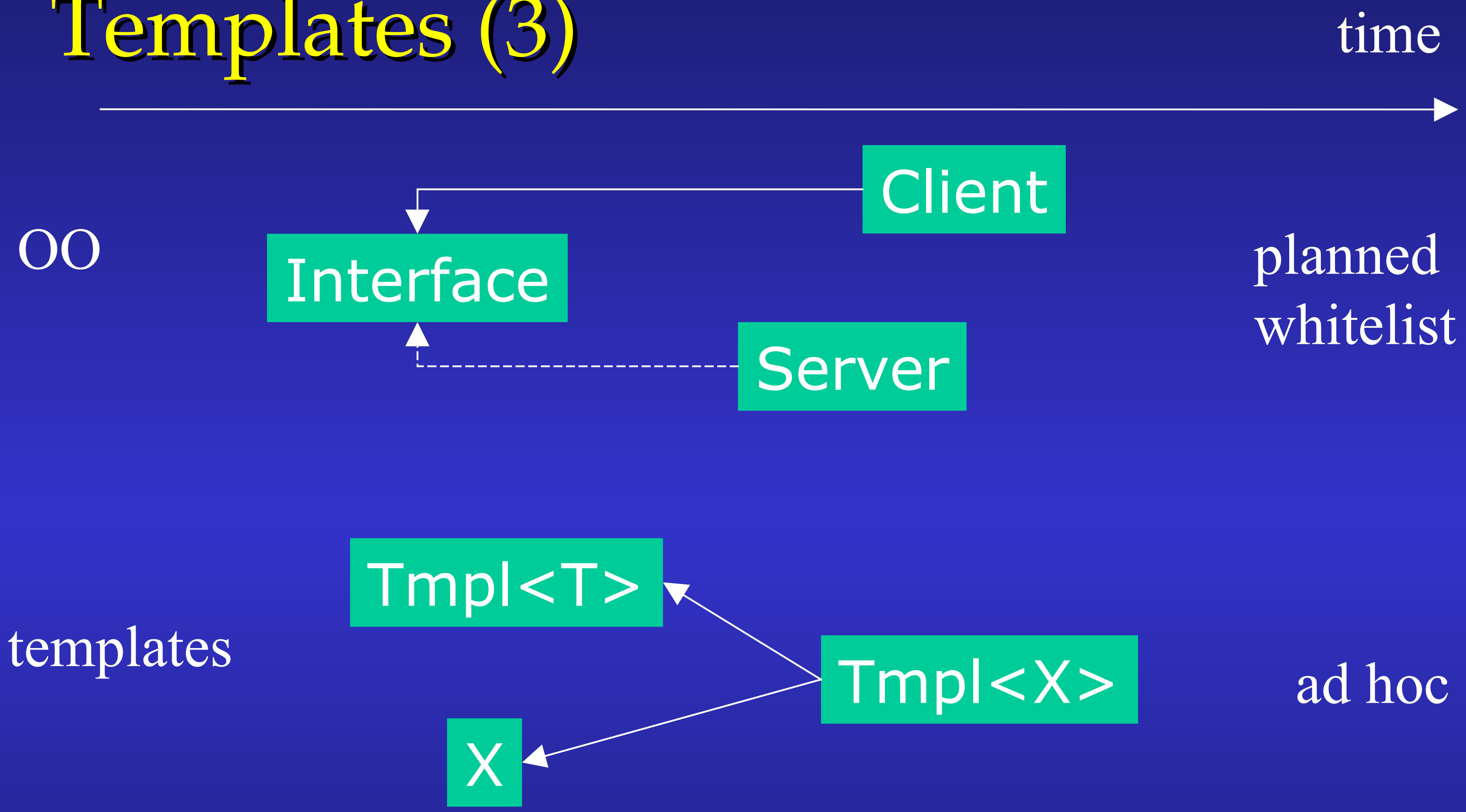
- Way more powerful than generics in other languages
- Source of endless hours of entertainment (and error messages)
- Essentially an accidental programming language within C++
- Two distinguishing features:
  - type deduction of function arguments
  - specialisation of templates



Templates (2)

- Type deduction of function arguments
  - e.g. make_pair(5, “foo”), sort(v.begin(), v.end())
  - allows for generic code without having to provide explicit types (no < or > required)
- Specialisation of templates
  - allows for different implementations for different argument types
  - compile-time polymorphism
  - traits, type-based lookup tables
  - optimised algorithms (std::distance)

Templates (3)



- Templates can combine things from different worlds
- Lack of interface contract leads to horrid error messages
- Optimisation possibilities are higher (inlining)

What makes C++11/14 different?

value semantics	➔	move and in-place semantics explicit conversions, literals
destructors	➔	exception safety and move ops
const	➔	constexpr
references	➔	“2 1/2” types of references
templates type deduction	➔	variadic templates, auto + decltype, “concepts lite”
lambdas	➔	little lambdas and big lambdas
libraries	➔	smart pointers, concurrency

Taking advantage of move semantics

```
class X {
public:
  // no special functions other than c/trs
private:
  int i = 0;                   // move == copy
  std::string s;               // moveable and copyable
  std::vector<int> v;          // similarly
  std::unique_ptr<int> p;  // only moveable
};
```

1. Get out of the way of the compiler
   - don't define copy or move operations and the compiler will generates move operations for you
   - use moveable or copyable types (i.e. not pointers)

Taking advantage of move semantics

```
X createAnX();
void f(X x);

f(createAnX());      // allows the temporary X to be moved
                     // if X is move-enabled
```

2. Consider pass/return-by-value for handle-based types
   - caller can optimise the copy
   - PGO/WPO may do this for you anyway
   - return-value optimisation (RVO) may even eliminate (elide) the move completely

Taking advantage of move semantics

```
// instead of repetitive overloading on lvalue/rvalue:
//  void f(const X & x) { v.push_back(x); }       // copy
//  void f(X && x) { v.push_back(std::move(x));  } // move

template <typename T>
void f(T && x) { v.push_back(std::forward<T>(x)); }
```

3. If move really is an optimisation, use a template with std::forward to preserve lvalue/rvalue-ness instead of having both const X & and X && versions
   - "deep move" requires doing this down the whole call stack

Move and std::swap

```
template <typename T>
void std::swap(T & a, T & b)
        noexcept(/* move ops don't throw */)
{
   T temp = std::move(a);
   a = std::move(b);
   b = std::move(temp);
}
```

- Exception-safe code relies on nothrow swap
- If move ops can throw, std::swap can throw
  - Move operations that can throw are dangerous!
- Replace member swap with std::swap for nothrow movable types

Move-only types

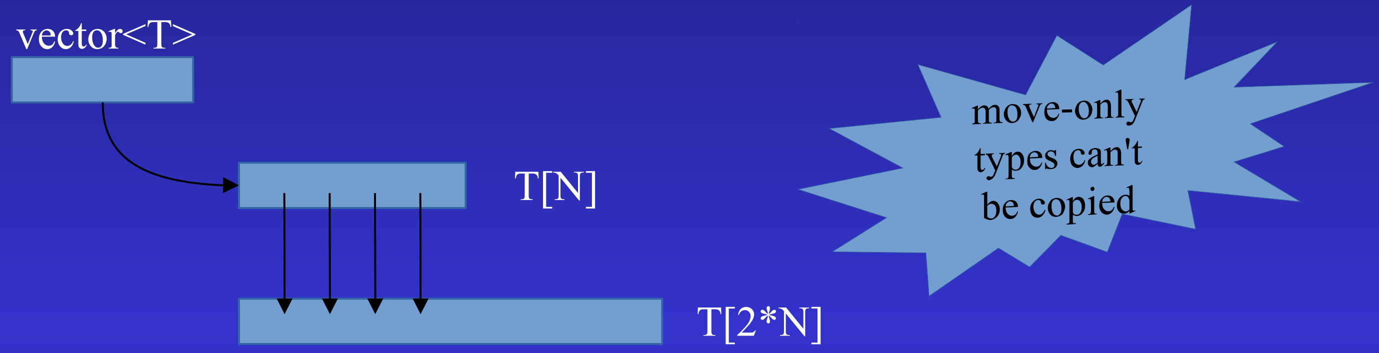
```
std::vector<Thread *> vtp;      // C++98/03

std::vector<std::thread> vt;    // C++11
```

- Introduction of move-only types (std::thread, std::unique_ptr, std::ostream, etc) simplifies object lifetime management
- Raises issues of what a move-from object should contain
  – not a problem for “move as optimisation” as robbed-from temporaries aren't visible

Move and std::vector

- std::vector::push_back may have to reallocate



<u>"happy day" scenario</u>
1) allocate T[2*N]
2) copy Ts
3) point to new array
4) deallocate T[N]

commit

<u>copy error</u>
1) allocate T[2*N]
2) copy Ts (bang!)
3) deallocate T[N]

rollback

<u>move error</u>
1) allocate T[2*N]
2) move Ts (bang!)
3) unmove???

help!?

- Move ops that throw make std::vector copy instead
- Move is a destructive operation with no undo

Move and non-move variants

```
struct X { X(X &&) noexcept(false) {} };

template <typename T>
struct stack {
    template <typename U=T>
    typename std::enable_if<std::is_nothrow_move_constructible<U>::value, U>::type pop()
    { return U(); }

    template <typename U=T>
    typename std::enable_if<! std::is_nothrow_move_constructible<U>::value, void>::type pop()
    {}
};

int main() {
    stack<int> si;
    stack<X> sx;

    int i = si.pop();
    //X x = sx.pop();          // compilation error
    sx.pop();
}
```

```
// not strongly ES
T stack<T>::pop() {
  return data[--index];
}
```

- Return by value can be made exception safe using traits

Beyond move – in-place semantics

```
// instead of copying/moving into a container, get the
// container to create the object for you in-place
//
// v.push_back(x);

v.emplace_back(param1, param2);   // pass params to X's c/tr
```

- C++11 containers can create elements in-place, avoiding the copy/move completely
  - RVO → move → copy for return values
  - emplace → move → copy for containers
  - caller optimisation → move → copy for inputs
- Optimisation hierarchy (emplace, move, copy)

Constexpr – const on steroids

```
template <int N> // C++03
struct Fact {
  static const int result =
    N * Fact<N-1>::result;
};

template <>
struct Fact<1> {
  static const int result = 1;
};

int array[Fact<5>::result];
```

```
constexpr int fact(int n) { // C++11
  return n == 1 ? 1  : n * fact(n-1);
}
// can have only a return statement
```

```
constexpr int fact(int n) { // C++14
  int acc = 1;
  for (int i = 2; i < n; ++i) acc *= n;
  return acc;
}

int array[fact(5)];
```

- Can apply to functions, c/trs and literals
- C++14 extends syntax and lifts some restrictions
  - can replace compile-time calculations done using template metaprogramming or macros

Constexpr – const on steroids

```
struct Kilo {
  constexpr Kilo(double d) : d{d}
  double d;
};

constexpr Kilo operator"" _kg(long double d) { return Kilo(d); }

auto k = 3.45_kg;
```

- Can now create compile-time constants of user-defined types
  - type safety
  - all conversions done and checked at compile-time
  - no temporaries
  - no multi-threading race conditions

Constexpr – const on steroids

```
enum class UnitType { kiloType, metreType };

template <UnitType U>
struct Unit {
    explicit constexpr Unit(double u) : u{u} {}
    const double u;
};

using Kilo = Unit<UnitType::kiloType>;       // equivalent of typedef
using Metre = Unit<UnitType::metreType>;

constexpr Kilo operator"" _kg(long double d) { return Kilo(d); }
constexpr Metre operator"" _m(long double d) { return Metre(d); }
```

```
auto kg = 3.4_kg;
auto m = 5.6_m;
auto d = kg + m; // oops!
    // compiler error
```

- "Type-rich interfaces" for clearer and safer APIs
  - allows for non-aliased simple types (c.f. Pascal):
    void sleepFor(Milliseconds ms);
- C++14 has literal suffixes for complex (1.2 + 3.4i)

Constexpr – compile-time fun

- In C++98/03 there is no way to go from a run-time value to its type other than by function template argument deduction
- In C++11 decltype can do this (mostly the same)
- Can also be done at compile-time with constexpr
- Allows for a unification of expression with the two languages: templates and run-time language
- C++14's extensions to constexpr make this much more useful and appealing

```
constexpr int f() { return 5; }

std::array<decltype(f()), f()> arr;       // std::array<int, 5> arr;
```

Constexpr – unit testing

```
// the world's smallest unit-testing framework?
#define  CCHECK(x)  static_assert((x), #x)

struct X {
    X(X &&) noexcept {}
    static void check() {
        CCHECK(std::is_nothrow_move_constructible<X>::value);
    }
};
```

- We can (and should) unit test constexpr code
- Can be done at compile time
- Checks expectations on use and i/f contract
- Zero overhead on run-time size or speed (assuming a good linker)

Constexpr – unit testing

```
// the world's smallest unit-testing framework?
#define  CCHECK(x)  static_assert((x), #x)
```

	Google Test	CCHECK
Test runner	separate download or library	compiler
Running tests	optional	mandatory
“Decapitate” main	required	not required
Isolate unit	dependency mgmt	not required
File/line reporting	__FILE__, __LINE__ macros	compiler
Test speed	can be slow	fast
Changes to build	required	none
Include header files	required	inline
Keep test code separate	required	inline

Constexpr – a possible future?

	constexpr code	non-constexpr code
constexpr context	✓	✗
non-constexpr context	✓	✓

compiler checks this already

- Since the compiler has to check if code is constexpr, why do we need to mark it as such explicitly? (c.f. auto)
- Should constexpr-ness be deduced by the compiler?
  - Should constexpr become implicit?
- Could the compiler do more at compile time if it is allowed to deduce constexpr?
- Is there ever a case where you wouldn't want constexpr code to be usable at run-time?

References – 2½ types, two syntaxes

```
void f(const X &);      // now called lvalue reference

void f(X &&);             // rvalue reference

template <typename T>
void f(T &&);             // “universal” reference
```

- Rvalue reference has different matching rules in a deduced context (template, auto, decltype)
  - Can bind to lvalues and rvalues!
  - A different syntax would have helped…

Smart pointers

```
// C++11
auto p = std::make_shared<X>(param1, param2);
auto upa = std::unique_ptr<int[]>(new int[4]);
auto fp = std::unique_ptr<FILE, decltype(&fclose)>(
  fopen("/etc/myapp.cnf", "r"), fclose);

// C++14
auto p2 = std::make_unique<X>(param1, param2);
```

- There's far less need to write new or delete
  - more efficient in terms of memory allocations
  - common idiom of variadic constructor
- Generalised to arrays and custom deletion

Smart pointers

```
class X {
public:
  void updateObject() {
    auto p = make_shared<const XBody>(*pImpl);
    p->x = ...;
    pImpl.reset(p);       // copy-on-write
  }
private:
  std::shared_ptr<const XBody> pImpl;
};
```

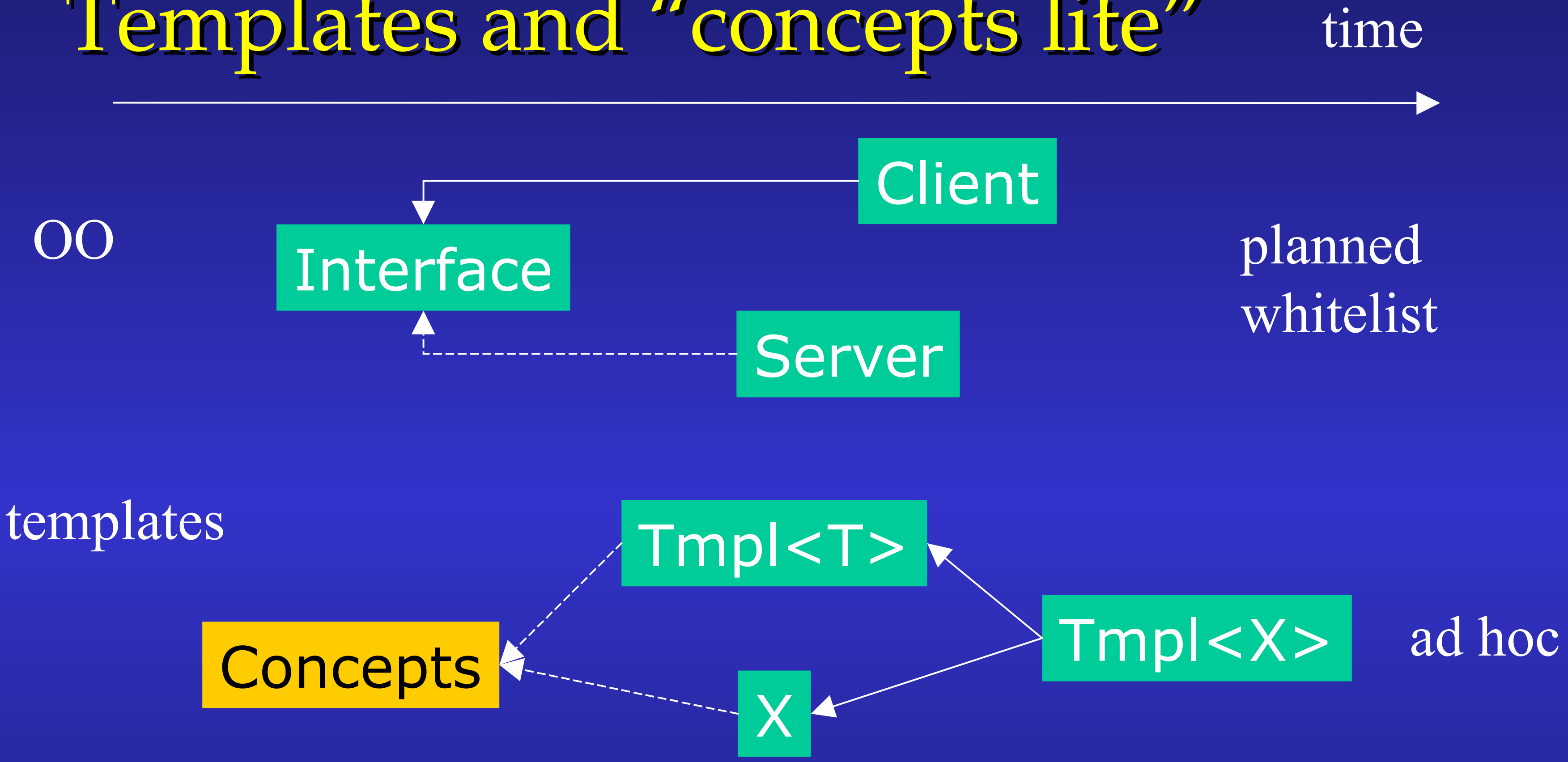
- Shared representation
  - generated copy/move ops are OK and efficient
  - immutable data easy to understand
  - appearance of deep copy but done efficiently
  - not thread safe

Variadic templates

```
class thread_guard {
public:
  template <typename... Args>
  thread_guard(Args &&... args) :
          t{std::forward<Args>(args)...} {}
  ~thread_guard() { if (t.joinable()) t.join(); }
private:
  std::thread t;
};
```

- Can pass multiple types in a typesafe way
- Commonly used for forwarding c/trs
  - pass arguments through using std::forward<T>
  - std::make_shared<X>(…), std::thread(…), etc

Templates and "concepts lite"



- "Concepts Lite" adds type checking to template arguments
- Better error messages, same or better compilation speed
- Aiming for C++17

Little lambdas

```
template <typename Func, typename... Args>
int execQuery(const std::string & sql, Func && func, Args && ... args) {
    PreparedStmt stmt(db, sql, std::forward<Args>(args)...);
    int counter = 0;
    while (sqlite3_step(stmt) == SQLITE_ROW) {
        counter++;
        func(stmt);
    }
    return counter;
}

db.execQuery("select * from users where username = :user",
        [](sqlite3_stmt * stmt) {
            for (int i = 0; i != sqlite3_column_count(stmt); ++i)
                cout << sqlite3_column_text(stmt, i) << " ";
            cout << endl;
        },
        username);
```

- Little lambdas very useful for refactoring loops
- Also with STL algorithms, handlers, inline code, etc...

Big lambdas

```
var server = net.createServer(function(socket) {            // node.js
  socket.addListener("connect", function() {
    sys.puts("Connection from " + socket.remoteAddress);
    socket.end("Hello World\n");
  });
});

$(document).ready(function() {  // JQuery
  $("button").click(function() {
    $("p").hide();
  });
});

// D3 visualisation library example – 4 levels of nested lambdas, 150 lines long

vector<future<void>> tasks;
tasks.push_back(std::async([]{ /* … */ }));     // C++11

auto f = std::async([]{ /*...*/ }).then([]{ /* ... */ });     // pass a continuation

if (f.ready()) { /* ... */ }                               // concurrency TS
when_any/all(future1, future2, ...);
executor.add([]{ /* ... */};
```

async completion code

async completion code

Type deduction – auto et al

```
auto j = 3;
decltype(j) k;                    // k has same type as j: int
for (auto i = 0; i != 10; ++i) {}
std::vector<X> vx;
for (auto x : vx) {}              // x is a copy
for (const auto & xr : vx) {}  // xr is a const lvalue reference
```

- Major convenience factor
- Don't forget the const and the reference…
  - particularly for move-only types!
- Makes C++ seem more like a dynamic language but with static compile-time type checking and performance

Type deduction, lambdas and functions

```
auto squared = [](int i){ return i * i; }   // returns int (C++11)

auto incr = [](auto i){ return i + 1; }
                  // returns decltype(i+1) in C++14
                  // effectively a templated lambda (but no <>!)

auto half(int i) { return i / 2; }
                  // returns decltype(i/2) in C++14
```

- Lambdas can deduce their return type
- Templates can deduce their argument types
- So why not mix the two together!
  - generic/polymorphic lambdas
  - deduced functions

So, where are we headed?

- Move semantics optimise value semantics
  - call/return by value will become more common and efficient (move and WPO)
  - exception safety issues unless move noexcept
  - move-only types are now easier (c.f. auto_ptr)
  - handle-based types with compact separate storage ideal for efficiency (e.g. std::vector, std::string), cache friendliness, vectorisation, etc
- Constexpr adds "third language" to run-time and template metaprogramming
  - zero overhead for separate types and literals
  - mixing of compile-time and run-time languages

So, where are we headed?

- C++ is starting to look much more like a dynamic language like JavaScript or Python
  - static compile-time checking with run-time efficiency and much reduced explicit typing
- Concurrency
  - task-oriented parallelism via big lambdas
    - not just object-oriented, procedural, generic, functional
    - asynchrony works well with concurrency TS
  - data parallelism using STL algorithms and lambdas
    - std::sort(policy, v.begin(), v.end());
    - policy == std::seq, std::par or std::vec
  - executors (thread pools, etc)