Using Modern C++

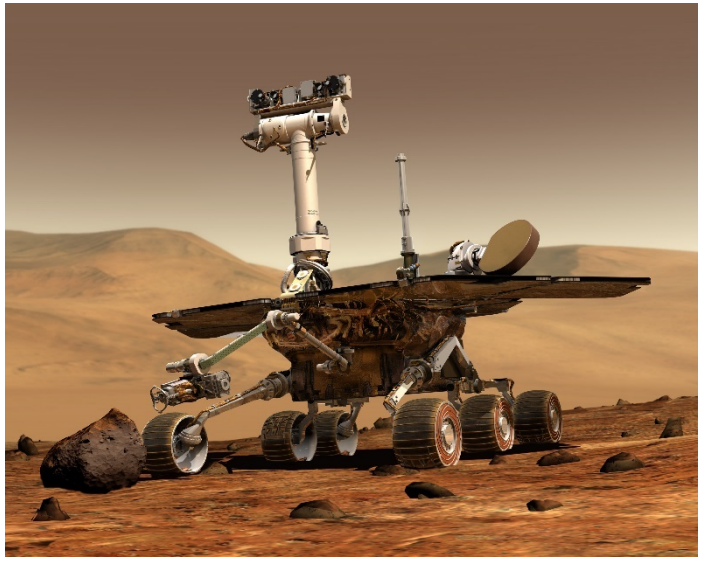
stepping up to C++17

Bjarne Stroustrup

Morgan Stanley, Columbia University

www.stroustrup.com

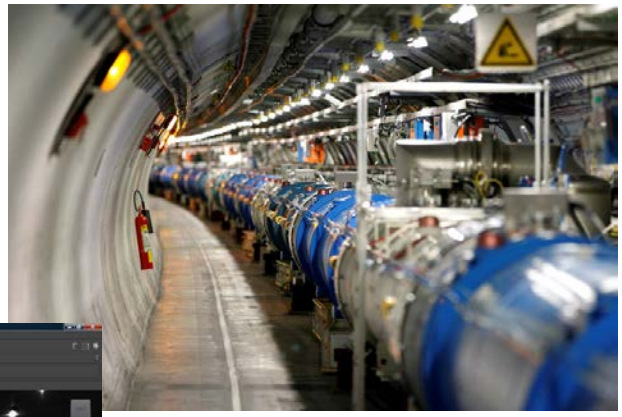
Overview



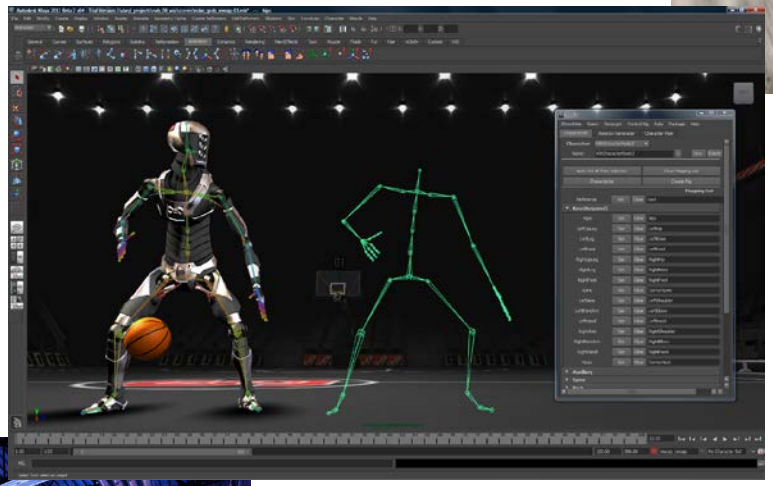
- Foundations
- Programming modern C++
  - Guidelines
    - Type-safety, resource-safety, simplification
  - Evolution
    - -> C++11 -> C++14 -> C++17 -> C++20 ->
  - Themes
    - Simplification, Compile-time computation, Programming as composition, …
- Feature lists
- Q&A

The value of a programming language is in the quality of its applications

Morgan Stanley



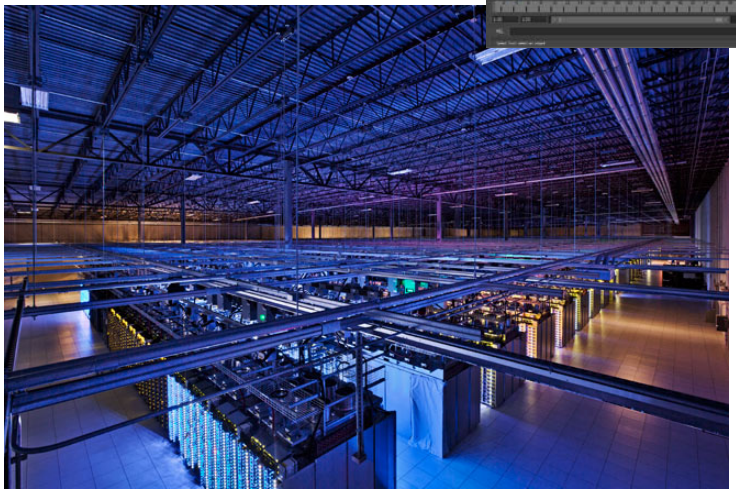
OpenOffice.org



Microsoft .net

Java

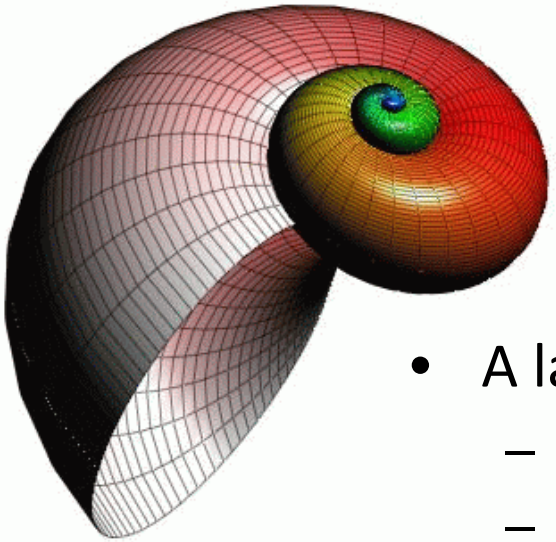
C#



amazon

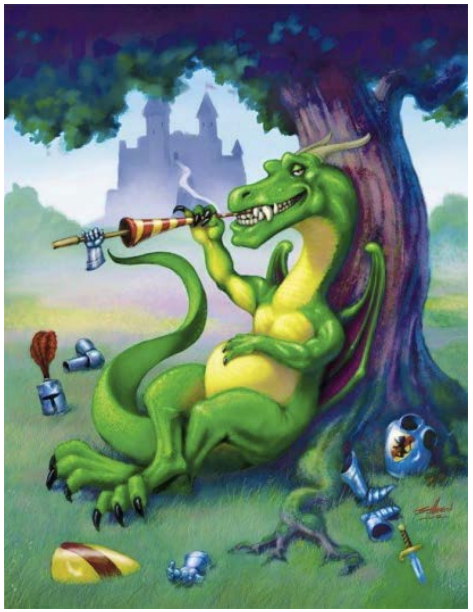
Google

AlphaGo



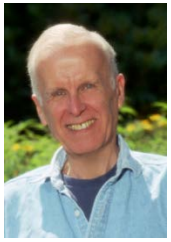
C++'s role

- A language for
  - Writing elegant and efficient programs
  - Defining and using light-weight abstractions
  - Resource-constrained applications
  - Building software infrastructure
- Offers
  - A direct map to hardware
  - Zero-overhead abstraction



- No language is perfect
  - For everything
  - For everybody

Programming Languages



Domain-specific abstraction

Fortran

Cobol

General-purpose abstraction

Simula

Java

C#

Direct mapping to hardware

C++

C++11

Assembler

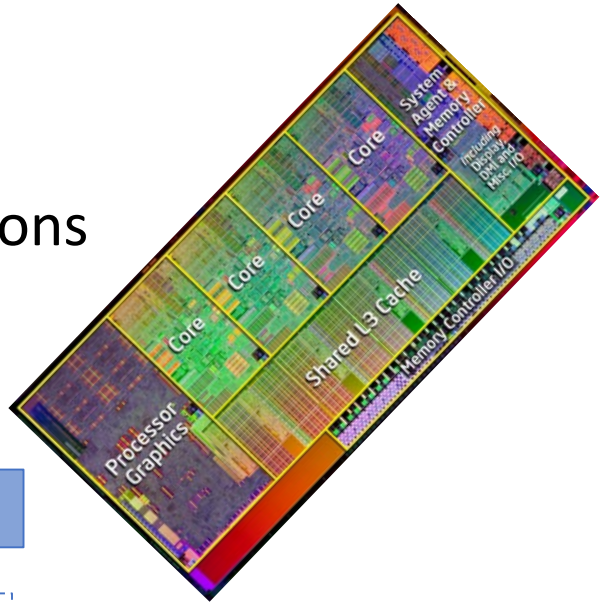
BCPL

C

C++14

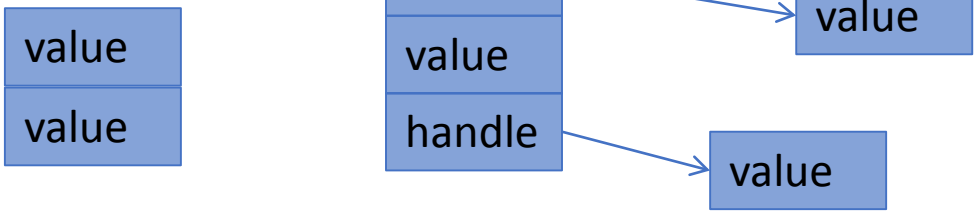
C++17

C/C++ machine model



- Primitive operations maps to machine instructions
  - +, %, ->, [], (), …
- Memory is a set of sequence of objects
  - Pointers are machine addresses

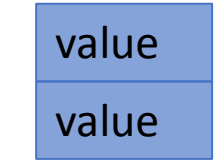
- Objects can be composed by simple concatenation
  - Arrays
  - Classes/structs

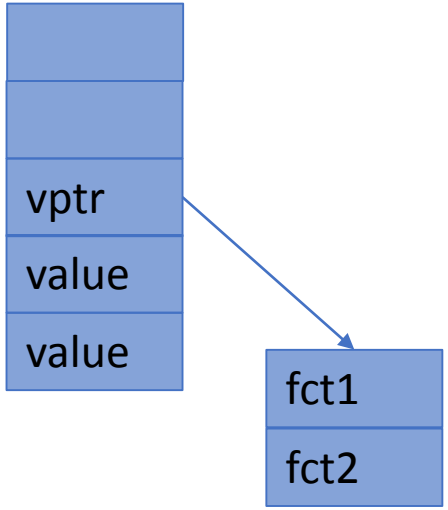


- The simplicity of this mapping is one key to C and C++'s success
  - It's a simple abstraction of hardware

Zero-overhead abstraction

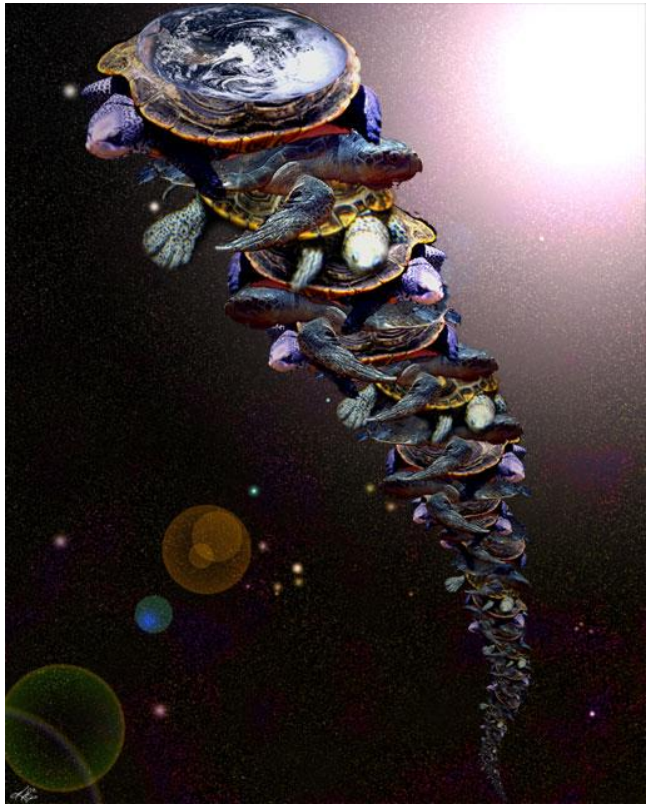
- What you don't use, you don't pay for
- What you do use, you couldn't hand code any better
  - So you can afford to use the language features
- Examples
  - Point, complex, date, tuple
    - No memory overhead
    - No indirect function call
    - No need to put on free store (heap)
    - Inlining
  - Compile-time computation
    - Pre-compute answers





It’s abstraction all the way down

- Abstraction
  - Can simplify understanding
  - Can simplify use
  - Can simplify optimization
  - Even our hardware is an abstraction



- Always abstract from concrete examples
  - Maintaining performance
    - Or you get bloat
  - Keep simple things simple
    - Or you get only “expert only” facilities

Resource management

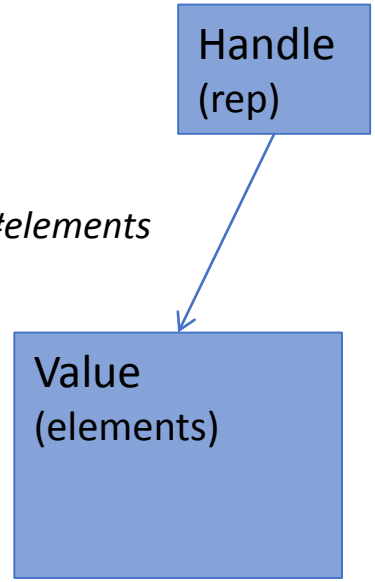
- A resource is something that must be acquired and released
  - Explicitly or implicitly
- Examples: memory, locks, file handles, sockets, thread handles

```
void f(int n, string name)
{
        vector<int> v(n);           // vector of n integers
        ifstream fs {name};         // open file <name> for reading
        // …
} // memory and file released here
```

- We must avoid manual resource management
  - No leaks!
  - Minimal resource retention

Constructors and destructors

```
template<Element T>
class Vector {          // vector of Elements of type T
public:
        Vector(initializer_list<T>);        // acquire memory for list elements and initialize
        ~Vector(); // destroy elements; release memory
        // …
private:
        T* elem;                // representation, e.g. pointer to elements plus #elements
        int sz;                 // #elements
};

void fct()
{
        Vector <double> v {1, 1.618, 3.14, 2.99e8};          // vector of 4 doubles
        Vector<string> vs {"Strachey", "Richards", "Ritchie"};
        // …
} // memory and strings released here
```



What is “modern C++”?

- “Modern C++” is ***effective use*** of ISO C++
  - Currently C++17 plus key ISO Technical Specifications
  - As the standard evolves, so does our notion of “modern C++”

- What is “effective use”?
  - Standards say what’s “legal” not what’s “good”/”effective”
  - Not just language
  - Tools
    - Compilers, analyzers, IDEs, profilers, test support, …
  - Techniques
    - Static type checking, Resource management, error handling, …
  - Libraries
    - Graphics, GUI, databases, networking, linear algebra, animation, …

C++ Core Guidelines

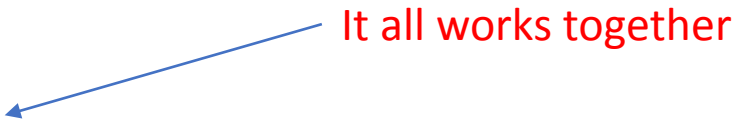
- You can write type- and resource-safe C++
  - No leaks
  - No memory corruption
  - No garbage collector
  - No limitation of expressibility
  - No performance degradation
  - ISO C++
  - Tool enforced (eventually)
- Work in progress
  - C++ Core Guidelines: https://github.com/isocpp/CppCoreGuidelines
  - GSL: Guidelines Support Library: https://github.com/microsoft/gsl
  - Static analysis support tools (work in progress)

C++ Core Guidelines

- What is modern C++?
  - A practical answer
- Aims
  - Far fewer bugs
  - Significantly simpler code
  - Uncompromising performance
- What features to use, and how
  - Coding rules
    - General
    - Domain specific
    - Use modern C++
  - Library
    - Parts of the standard library
    - GSL (tiny)
  - Static analysis

Today: C++11 -> C++14 -> C++17 ->

- What is availably for use now?
  - C++17
- What should we plan for next year?
  - Upgrade
  - Experiment
- Themes
  - Simplification
  - Compile-time computation
  - Programming as composition
  - Type-safety
  - Concurrency and Parallelism
  - Libraries
- Note the emphasis
  - What problems are addressed
  - Not on details of how to do it



Theme: Simplification

- What is simple?
  - For developers of quality systems?
- Not the lowest level facilities
  - **void*** (raw memory)
  - **goto**
  - Zero terminated arrays of characters
  - “Object”
  - Unconstrained generics (templates)
  - …
- With those, you can do anything
  - Incl. any mistake and any unmaintainable mess
- C++ must not be ***just*** expert friendly
  - Express ideas directly in code
  - Keep simple things simple
    - Don’t make complicated things impossible

Writing a loop

- 1972

```
int i;
// i available here, but uninitialized
for (i=0; i<max; i++) v[i]=0;
// i valid here (often unwanted)
```

- 1983

```
for (int i=0; i<max; ++i) v[i]=0;
```

- 2011

```
for (auto& x : v) x=0;
```

Keep simple things simple!

- Deprive bugs of their hiding places!
  - Bugs hide in complex code

- The simpler code is as fast as the old
  - and safer
  - **for (i=0; i<=max; j++) v[i]=0;** **//** *Ouch! And double Ouch!!*

Dangling pointers – my most dreaded bug

- One nasty variant of the problem


```
void f(X* p)
{
        // …
        delete p;    // looks innocent enough
}

X* q = new X;    // looks innocent enough
f(q);
// … do a lot of work here …
q->use();        // Ouch! Read/scramble random memory
```

- Never let a pointer outlive the object it points to
  - Easy to say, hard to do at scale

Dangling pointers – my most dreaded bug

- A focus of Core Guidelines
  - Static analysis support in the works
- Rely on scoped objects
- Move semantics:
  - We can return “large objects” cheaply
  - We can cheaply move “large objects” from scope to scope
- Encapsulate **new** and **delete**
  - A **delete** in application code is a bug waiting to happen
- Pointers are great for pointing when they don’t dangle
  - Distinguish owning pointers from non-owners; e.g., **gsl::owner<T*>**
- If you need owning pointers, make them smart
  - **shared_ptr** and **unique_ptr;** but don’t overuse

Move semantics – Eliminate "pointer fiddling"

- 1984 (pointers everywhere)

```
X* make_X(int arg) { X* p = new X(arg);  …  return p; }
X* p = make_X(4);
// …
delete p;       // easily forgotten
```

- 1998 (smart pointers – in too many places)

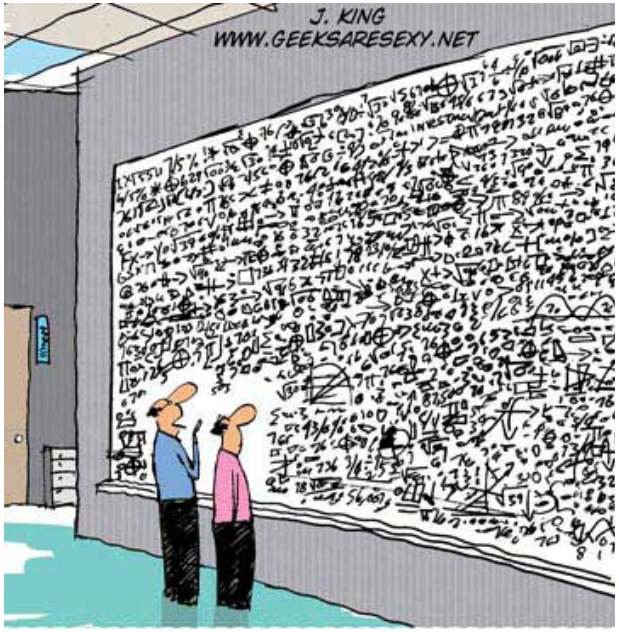
```
// …
auto_ptr<X> p(make_X(4));                    // later: shared_ptr ,unique_ptr
```

- 2011 (move semantics – no pointers)

```
X make_X(int arg) { X x {arg};  …  return x; }        // return "by value"
// …
auto x = make_X(4);

X x = [](){ X x {4};  … return x; }();                // lambda as initializer
```

Simplification

- There is so much we can do to simplify!



"...And that, in simple terms, is what's wrong with your software design."

- Don't try to simplify by adding lots or random improvements
  - Make things more regular
  - Make simple things simple
  - Don't try to make inherently complex things simple
  - Don't try to force everything into a single mold
  - Don't simplify at the cost of generality

Theme: Compile-time computation

- If you can find the answer at compile time
  - No run-time error handling
  - No race conditions
  - Faster code
    - Most often (not always)
  - Smaller object code
    - Often
- If you are not careful
  - Abysmal compile times
  - Difficult debugging
  - Poor maintainability
  - Code bloat (e.g., excess code replication)
- It is not my ideal to do everything at compile time

Constant expressions (C++11, C++14, C++17)

- **constexpr** brings type-rich programming to compile time
  - If you know the answer, just use it
  - It's hard to run faster than a table lookup
  - You can't have a race condition on a constant
  - Macros or template metaprogramming can be very error-prone

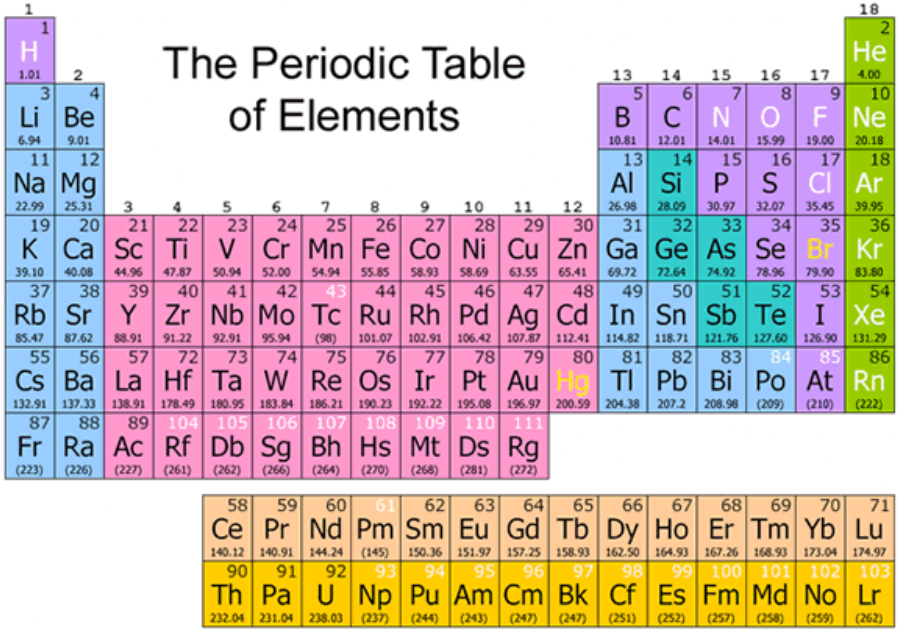
```
constexpr int isqrt(int n) // evaluate at compile time for constant arguments
{
        int i = 1;
        while (i*i<n) ++i;
        return i-(i*i!=n);
}

constexpr int s1 = isqrt(9);       // s1 is 3
constexpr int s2 = isqrt(1234); // s2 is 35

cout << weekday{oct/19/2017} << '\n';          // Thursday
static_assert( weekday{oct/19/2017}==thu );    // at compile time
```

User-defined literals (C++11, C++14)

- We have a small zoo of suffixes
  - **1u**, **1Ul**, **1.0l**, **1LL**, **1.0f, 1ull** **//** for literals of built-in types
- People can define similar suffices for their own types, e.g.
  - **123456789999991121323x** **//** Bigint
  - **12s, 17.3s, 12ms** **//** seconds, milliseconds
  - **1010101_10** **//** binary int ☺
  - **12min, 12h, 12day** **//** minutes, hours, days
  - **1m, 1s, 1kg** **//** physical units
  - **"a standard-library string"s** **//** a std::string
- No run-time overhead
  - **constexpr Second operator"" s(long double d) { return Second(d); }**
- Support user-defined types as well as built-in types are
  - Many don’t consider the constructor notation good enough

Theme: Composition

- Compose programs out of well-specified parts
  - Direct expression of ideas
  - Frameworks are too inflexible
    - Constraining
    - Adds overhead
    - Too general for every use
- Language support
  - Lambda expressions
  - Variadic templates
  - Optional and variant
  - Type and template aliases
  - Concepts
  - Modules



Lambda expressions

- A “lambda” is an unnamed function object
  - Avoid repetition of type
  - Preserve inlining opportunities
  - Improve locality

```
sort(v, [](auto& x, auto& y) { return x>y; });    // specify sorting criterion
```

- No, you don’t have to use them everywhere
  - every good new feature will be overused and misused

Lambda expression: callbacks

- 2001: binders

```
void doWork(InputMessage&& msg, TcpChannel chan)
{
    chan.send(msg);          // echo back
}

// … dozens of lines …

channel.setMessageCallback(
    bind(doWork, placeholders::_1, ref(channel))
);
```

- 2011: lambda

```
channel.setMessageCallback(
    [&channel](InputMessage&& msg) { channel.send(msg); }       // echo back
);
```

Variadic templates

- Handle any number of arguments of any types

```
template<typename… T>
void print(T&&… args)              // && means “forwarding reference”
{
      (cout<< … << args) << '\n’; // fold expression: start with “cout<<“
                                  //  then “pass each arg in args to cout using <<”
}

print("Hello"s, ' ', "World ",2017);      // string, char, const char*, int
print(x,y,z);                             // for any type that has a <<
```

Variadic templates

- Useful for forwarding
  - Note: no artificial bundles of parameters needed for InputChannel
  - Works for a variety of Transport types and constructors

```
template<typename Transport, typename MessageAdapter>
class InputChannel {
public:
    // …
    template<typename... TransportArgs>
        InputChannel(TransportArgs&&... transportArgs)
                : _transport {std::forward<TransportArgs>(transportArgs)... }
        {}
    // …
   Transport _transport;
 };
```

Type and template aliases

- Aliases as glue among abstractions
- Feature composition

```
template<typename E, typename A>
struct Policy {
        using Executor = E;          // "associated types"
        using Allocator = A;
};

Policy<Sequential,Default_allocator> seq;        // define a policy
```

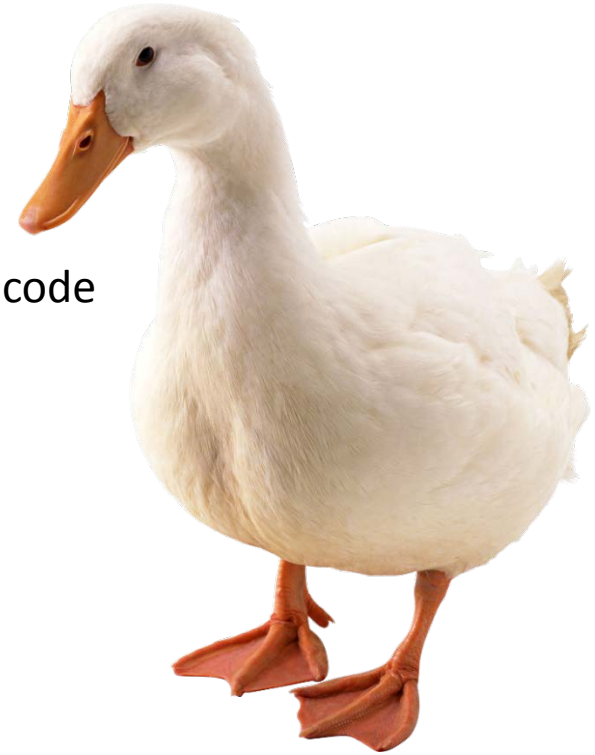
- Notational simplification

```
template<typename T>
using My_policy = Policy<My_executor<T>,My_allocator<T>>;

My_policy<Quad> Qpol;        // define a policy for manipulating Quads
```

Generic Programming

- 1980: Use macros to express generic types and functions
- 1987 (and current) aims:
  - Extremely general/flexible
    - "must be able to do much more than I can imagine"
  - Zero-overhead
    - **vector**/**Matrix**/… to compete with C arrays
  - Well-specified interfaces
    - Implying overloading, good error messages, and maybe separate compilation
- "two out of three ain't bad"
  - But it isn't really good either ☹
  - It has kept me concerned/working for 20+ years
  - Concepts! (now available)

Generic Programming



- Problem:
  - Templates offer compile-time duck typing
    - Encourages write-only, complex, unmaintainable code
    - Delivers appalling compiler error messages
  - Templates are useful and wildly popular
    - Flexible
    - Unsurpassed performance
    - Metaprogramming
    - Compile-time computation
- Solution
  - Generalized constant expressions: **constexpr** (C++11, …)
  - Precisely specified flexible interfaces: **concept**s (C++20)

Concepts: use

- What if we do want to sort vectors and lists?

```
void sort(Sortable& container);    // sort something that is Sortable

void sort(List& seq)     // sort something without random access
{
        vector<Value_type<List>> v {begin(seq),end(seq)};
        sort(v);
        copy(begin(v),end(v),seq);
}

sort(vec);       // OK: use sort of Sortable
sort(lst);       // OK: use sort of List
```

- We don't say something like "**List < Sortable**"
  - We compute that from their definitions

Concepts: definition

- A concept is a compile-time predicate
  - On a set of types and values
  - Specified as use patterns

```
template<typename T>
concept Sequence = requires(T a) {
        typename T::Iterator;       // must have an iterator type
        { begin(a) }->Iterator;     // has a beginning
        { end(a) } -> Iterator;     // has an end
        Input_iterator<T::Iterator>;  // the iterator must be an Input_iterator
}

template<typename T>
concept Sortable = Sequence<T> &&
                   Random_access<T> &&      // has [], +, etc.
                   Ordered<Value_type<T>>;  // has <, etc.
```

Concepts

- Now in the standard's Working Paper scheduled for C++20
  - Except the "natural notation"

    void sort(Sortable&)
  - Some people prefer to have only the "shorthand notation"

    template<Sortable S>

    void sort(S&);
  - Or even just

    template<typename S>

    requires Sortable<S>

    void sort(S&);

- Available in GCC 6 and newer

Concepts: a real-world example

```
template <typename T>
concept InputTransport = AsyncObject<T> &&
      requires(T t) {
            typename T::InputBuffer;
            typename T::MessageCallback;
                  { t.setMessageCallback(typename T::MessageCallback{})  };
    };

template <typename T>
concept OutputTransport = AsyncObject<T> &&
            requires(T t, ::asio::const_buffer b) {
                  { t.send(b) };
            };
```

Real-world composition

```
template<typename Transport, typename MessageAdapter>
   requires concepts::InputTransport<Transport> &&
            concepts::MessageDecoder<MessageAdapter>
class InputChannel {
public:
    using InputMessage = typename MessageAdapter::template
                         InputMessage<typename Transport::InputBuffer>;
    using MessageCallback = typename std::function<void(InputMessage&&)>;
    using ErrorCallback = std::function<void(const std::error_code&)>;

    template<typename... TransportArgs>
    InputChannel(TransportArgs&&... transportArgs)
          : _transport(std::forward<TransportArgs>(transportArgs)...)
    {}
   // ...
   Transport _transport;
};
```

Actions: often lambdas

Associated types: aliases

Requirements: concepts

Variadic template

Forwarding

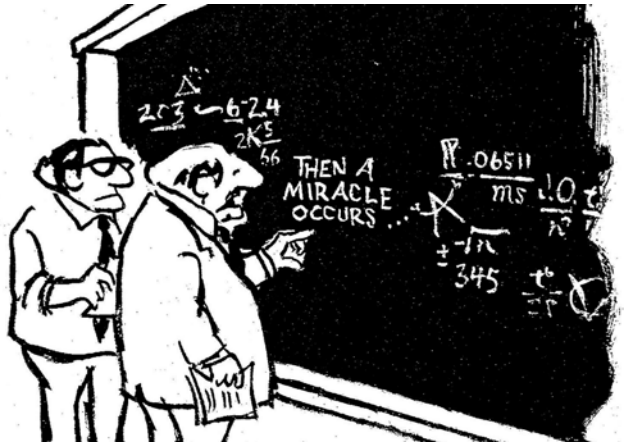
Modules

- Better code hygiene: modularity (especially protection from macros)
- Faster compile times (hopefully factors rather than percent)

```
import iostream;                    // use iostream
using namespace std;

module map_printer;                 // we are defining a module

export template<Sequence S>
        requires Printable<Key_type<S>> && Printable<Value_type<S>>
void print_map(const S& m)
{
        for (const auto& [key,val] : m)        // break out key and value
                cout << key << " -> " << val << '\n';
}
```

Modules

- An ISO Technical Specification
  - Some opposition
    - Some related to lack of macros support
    - Some worry about difficult type of transition from #include messes
    - Convergence of designs
  - You don't get factors of compile-time speed improvement
    - Without effort (not much)
    - Without improved code hygiene (great in itself)
- Implementations
  - Shipping in recent Microsoft compilers
  - Implementation in GCC has started
  - Somewhat different design in Clang

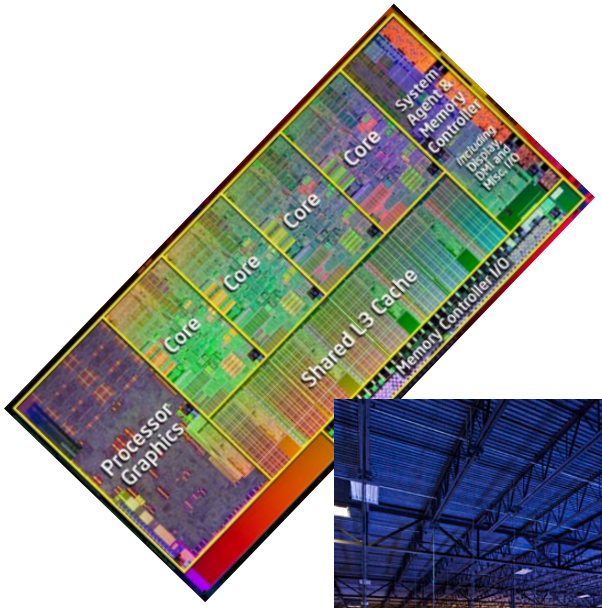


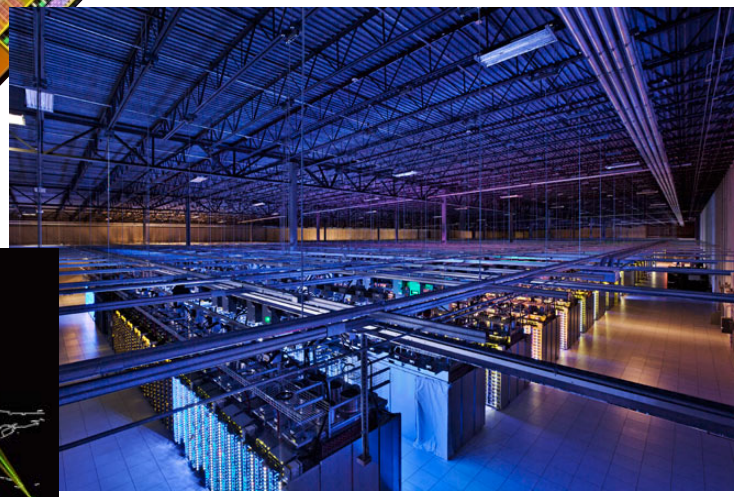
But does it scale?

- Yes
  - The small examples and techniques shown are the bases on enterprise-scale applications
  - Enable fast response to new use cases
  - Encourage experimentation
  - Allow application developers to decide what level of library they want to use
- Care needed
  - Concepts need to be well thought out
  - Focus and precision of definition are key
  - Don’t try to abstract too early
  - Don’t over-abstract

Theme: Concurrency and Parallelism

- Lock-free programming
- Threads and locks
- Futures
- Coroutines
- Parallel algorithms
- Vectorization
- Networking





Threads and locks (C++11)

- The worst way of writing concurrent and parallel programs
  - And the most common
    - Easy to get 99% right
  - The system/hardware level basis for higher-level models
  - **thread**, **mutex**, **condition_variable**, …

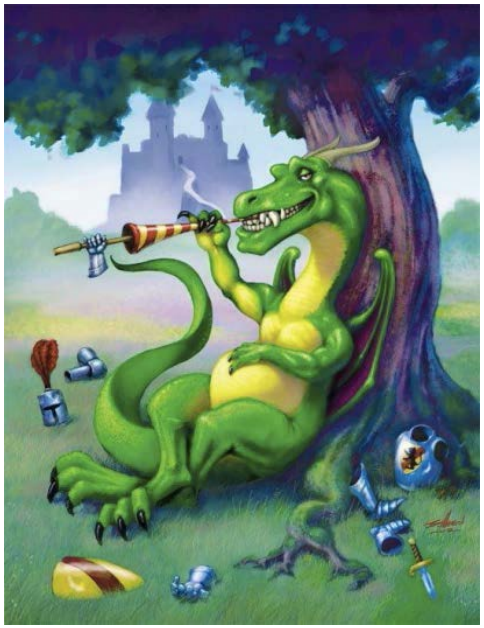
- C++20: latches and barriers
  - Helps in implementing parallel algorithms

Lock-free programming

- For people who really, really have to
- For people who find juggling sharp knives exciting

- Memory models
- Compare and swap primitives

- Soon: hazard pointers



Futures

- Pass a value to another thread
  - Without explicit locking

- C++11: Futures

```
future<Res> f= async([]() { return computation(x,y,z); });
// …
auto x = f.get();
```

- C++20: futures (already shipping in places)

```
future<string> g = f.then(
        [](future<Res> f2) { return f2.get().to_string(); }
                          );
```

Coroutines

- Stackless and stackfull

```
gen<int> fibonacci() // generate 0,1,1,2,3,5,8,13 …
{
        int a = 0;  // initial values
        int b = 1;

        while (true) {
                int next = a+b;
                co_yield a;          // return next Fibonacci number
                a = b;               // update values
                b = next;
        }
}

for (auto v: fibonacci()) cout << v << '\n';
```

- For coroutines and ranges see: http://ericniebler.com/2017/08/17/ranges-coroutines-and-react-early-musings-on-the-future-of-async-in-c/

Parallel algorithms

- We can add requests to parallelize and/or vectorize to most standard-library algorithms

```
// 23.19.7, execution policy objects
constexpr sequenced_policy seq { unspecified };
constexpr parallel_policy par { unspecified };
constexpr parallel_unsequenced_policy par_unseq { unspecified };

sort(v.begin(), v.end());                    // as ever
sort(seq, v.begin(), v.end());               // sequential sort, not parallel
sort(par, v.begin(), v.end());               // permit parallel
sort(par_unseq, v.begin(), v.end());         // permit  parallel and/or vectorize
```

- Soon
  - **vec** and **unseq**
- And I still want

```
sort(unseq, v);                              // Simplify!
```

Theme: Libraries

- Writing code in the bare language is tedious and error-prone
- With the right library, just about any task is easy
- Many/most modern C++ features support the design, implementation, and use of libraries
  - Classes, exceptions
  - Modules, concepts, variadic templates, structured binding, move semantics, lambdas
  - …

Libraries

- There are lots
  - Most are not standard
- C++17 standard-library components
  - String
  - I/O streams
  - File system
  - STL (containers and algorithms)
  - **<random>**, **<chrono>**, **<regex>**
  - Lock-free programming, threads and locks, futures, parallel algorithms
  - Simple numerics
  - Special math
  - Smart pointers (**shared_ptr** and **unique_ptr**)
  - Utilities (e.g. **variant** and **optional**)
- TSs
  - Ranges (STL2?)
  - Transactional memory
  - Networking (asio)

Conjecture

- The most direct expression of an idea is
  - The simplest to write
  - Statically type safe
  - The fastest
  - Most composable
  - Easiest to maintain
- Aim for that
  - Keep simple things simple!

C++11: "Feels like a new language"

- Concurrency support
  - Memory model
  - Atomics and lock-free programming
  - Threads, mutex, condition_variable, futures, …
- Move semantics
- Generalized constant expression evaluation (incl. **constexpr**)
- Lambdas
- **auto**
- Range-**for**
- **override** and **final**
- **=delete** and **=default**
- Uniform initialization (**{}** initializers)
- User-defined literals
- **nullptr**
- **static_assert**
- Variadic templates
- Raw string literals (**R"(…)"**)
- **long long**
- Member initializers
- …

- **shared_ptr** and **unique_ptr**
- Hash tables
- **<random>**
- **<chrono>**
- …

C++14: "Completes C++11"

- Function return type deductione.g. **auto f(T& x) { return g(x); }**
- Relaxed **constexpr** restrictions e.g. **for** in **constexpr** functions
- Variable templates
- Binary literals e.g., **0b0101000011111010**
- Digit separators e.g., **0b0101'0000'1111'1010**
- Generic lambdas e.g., **[](auto x) { return x+x; }**
- Lambda capture expressions
- **[[deprecated]]**
- …
- Shared mutexes
- User-defined literals for time
- Type-based tuple addressing e.g., **get<int>(t)**
- …

C++17: "a little bit for everyone"

- Structured bindings. E.g., **auto [re,im] = complex_algo(z);**
- Deduction of template arguments. E.g., **pair p {2, "Hello!"s};**
- More guaranteed order of evaluation. E.g., **m[0] = m.size();**
- Compile-time if, E.g., **if constexpr(f(x))** …
- Deduced type of *value* template argument. E.g., **template<auto T>** …
- **if** and **switch** with initializer. E.g., **if (X x = f(y); x)** …
- Dynamic memory allocation for over-aligned data
- **inline** variables (Yuck!)
- **[[fallthrough]]**, **[[nodiscard]]**, **[[maybe unused]]**
- Fold expressions for parameter packs. E.g., **auto sum = (args + …);**
- …
- File system library
- Parallelism library
- Special math functions. E.g., **riemann_zeta()**
- **variant**, **optional**, **any**
- **string_view**
- …

The future? (C++20)

- Language (major features)
  - Concepts (shipping: GCC; work in Microsoft)
  - Modules (shipping: Microsoft; work in Clang and GCC)
  - Contracts
  - Coroutines (shipping: Microsoft and Clang)
  - Static reflection
- Libraries
  - Networking (in widespread production use)
  - Better parallel algorithms
  - Ranges
  - 2-D graphics
- ***If*** we get those
  - and not “compromised or diluted”
  - C++20 will be ***GREAT***
  - C++20 will change the way we think about C++ programming
    - Like C++11 did

Questions?

- C++:
  - C++11, major
  - C++14, minor
  - C++17, minor
  - C++20, ???, promising