Introduction to modern C++

Olve Maudal

C++ has evolved a lot since it was first introduced as "C with classes" with primitive support for object-oriented programming. In particular during the last 10-15 years the common use of the language has changed "dramatically" and the language itself has evolved accordingly. Modern C++ (C++11/14) is still very suitable for object-oriented programming, but now the language also provides good support for generic programming and functional programming. All of this while C++ is still a low-level language that can be used to create programs that compete with programs written in assembler both in terms of speed and size.

We start with a brief history of C++ before focusing on new features in C++11/14 and a demonstration of some typical modern programming techniques.

a 3.5 hour workshop (including lunch)
Lysaker, Jan 29, 2015

- Brief History of C and C++
- New features in C++11/14
- Generic Programming
- The future of C++

- Evolution of Ski Jumping
- Brief History of C and C++
- New features in C++11/14
- Generic Programming
- The future of C++

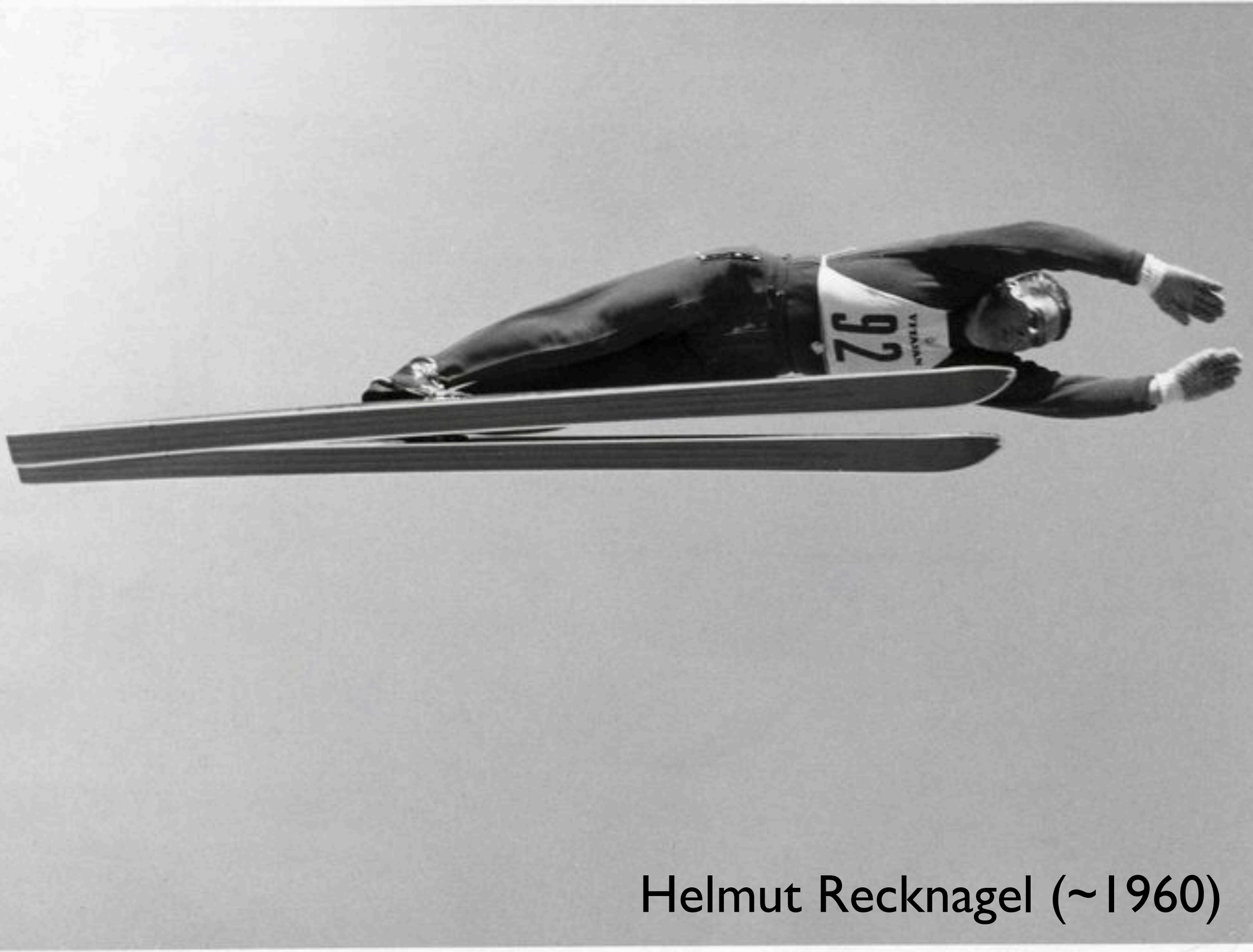
Evolution of Ski Jumping

http://www.aftenposten.no/fakta/innsikt/Da-skiene-ble-vinger-7459947.html

Opptrekk (1905)

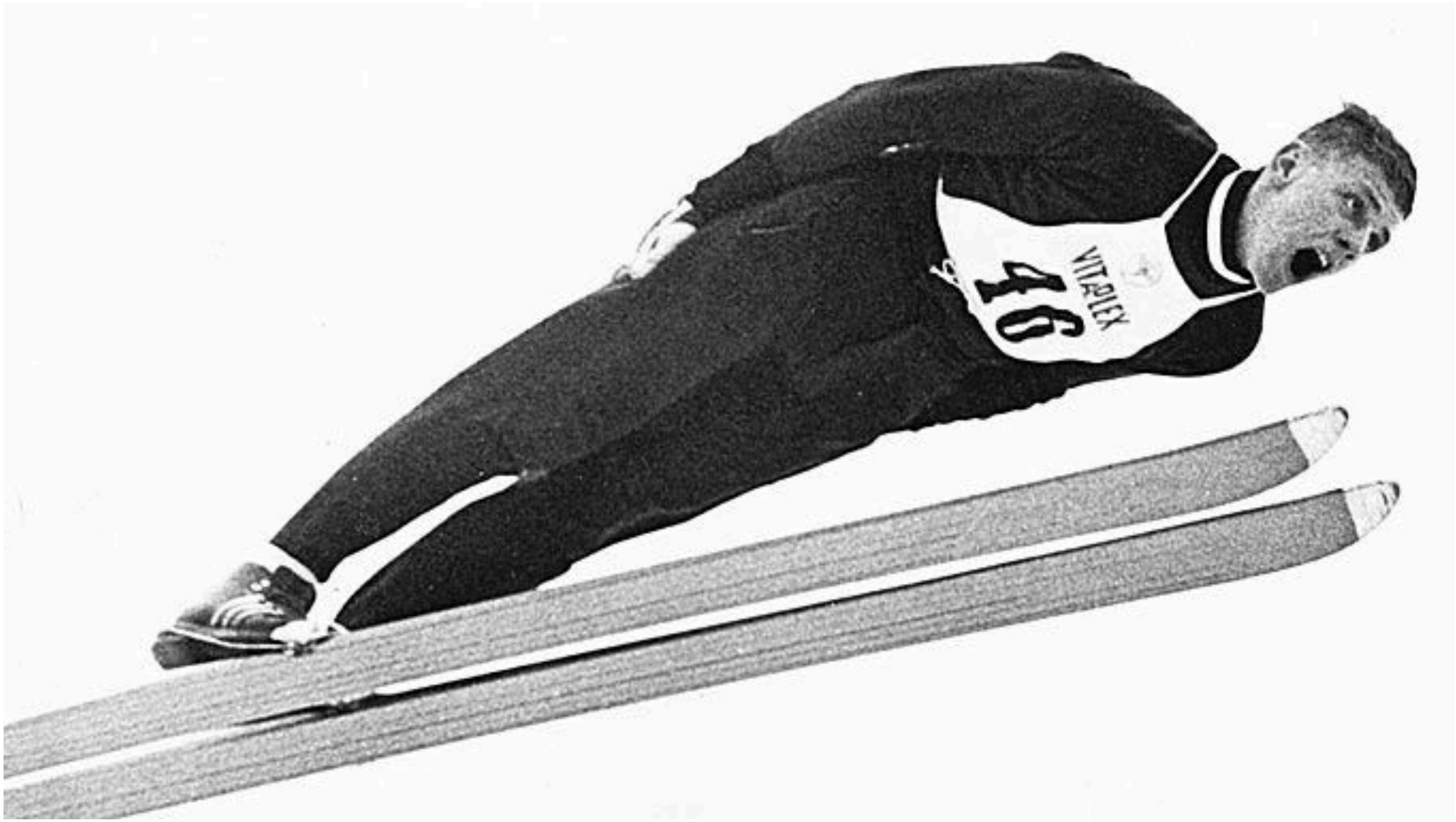
Kongsberg knekk, Birger Ruud (1947)

Matti Pietikäinen (1954)



Helmut Recknagel (~1960)

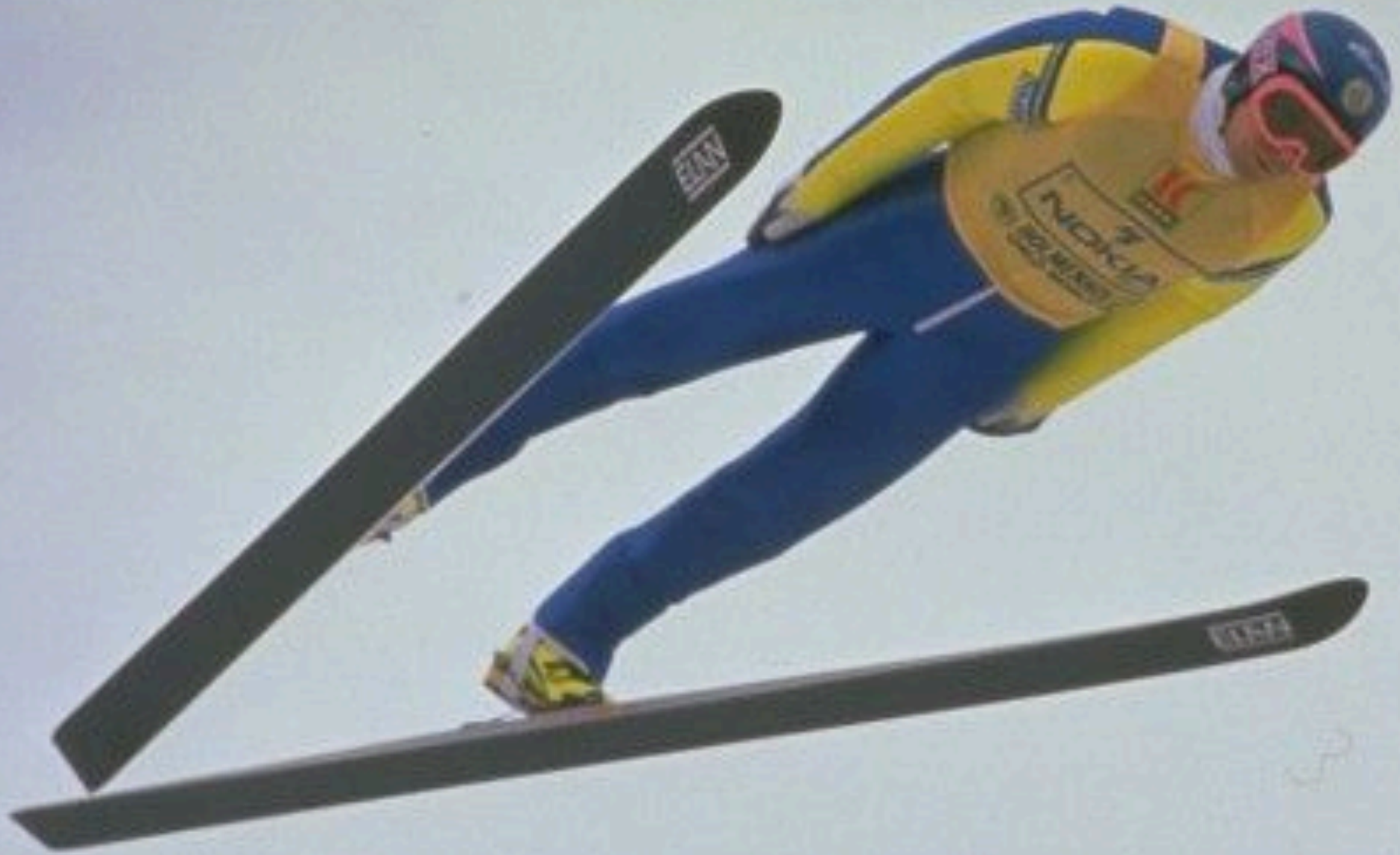
Finnestilen, Bjørn Wirkola (1964)



Finnestilen, Lars Grini (1967)

Sideflyt, Per Bergerud (1981)

Sideflyt, Per Bergerud (1983)



V-stilen, Jan Bokløv (1989)

Kamikaze, Noriaki Kasai (2004)

Dykkstil/W-stil, Andreas Wank (2012)

Brief History of C and C++

40's
Machine code, symbol tables and Assembler

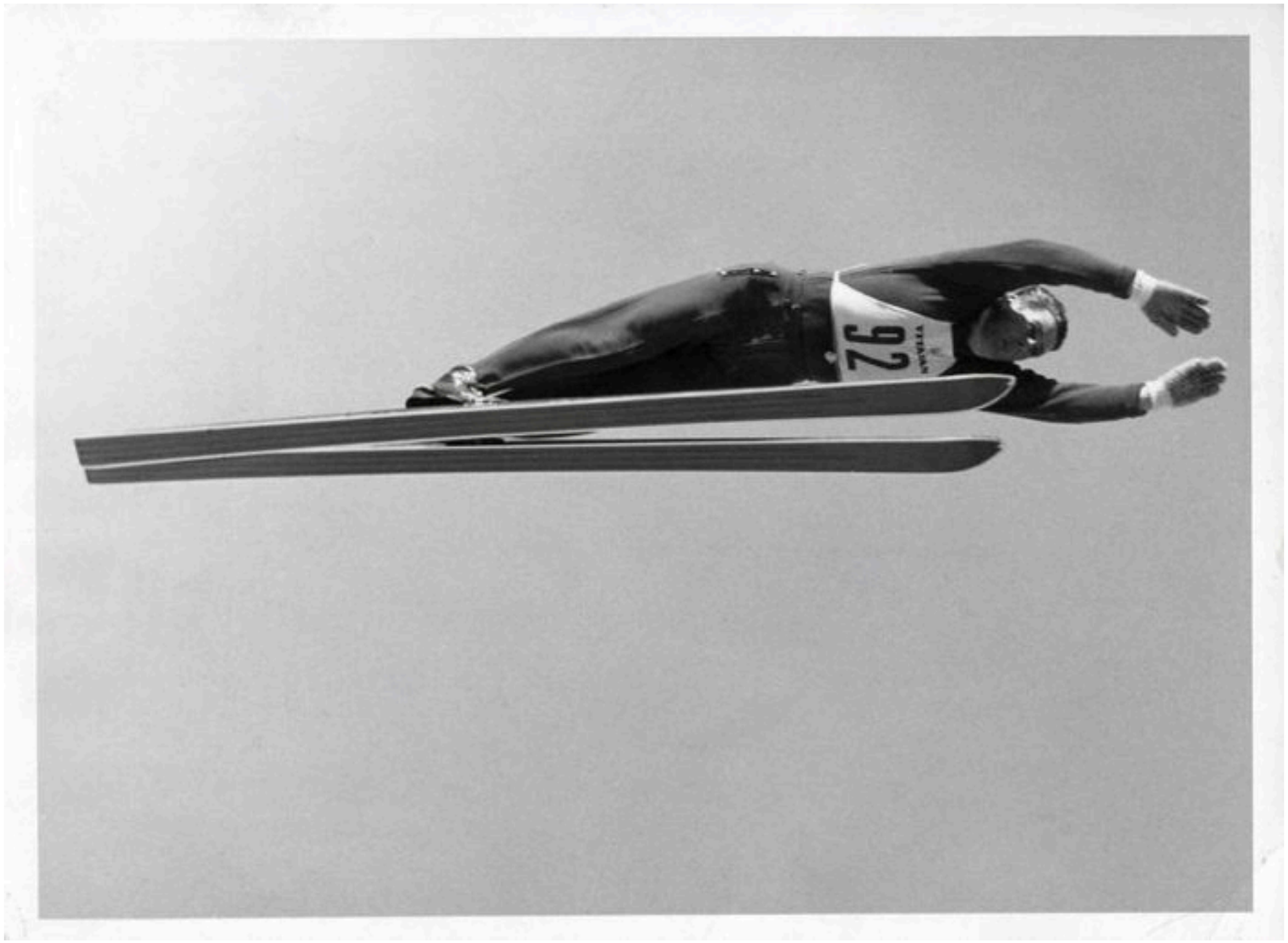
50's
Fortran, Lisp, Cobol, Algol

50's
Fortran, Lisp, Cobol, **Algol**

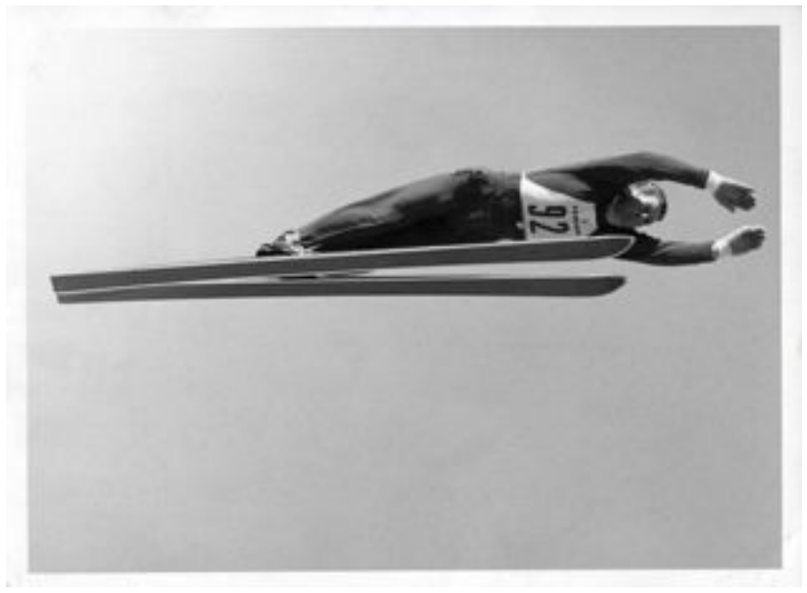
60's

many, many new languages appeared in this period. In particular...

CPL



60's
... and Simula



Both CPL and Simula were examples of **very** elegant languages...

... but there was also a need for brutally efficient languages

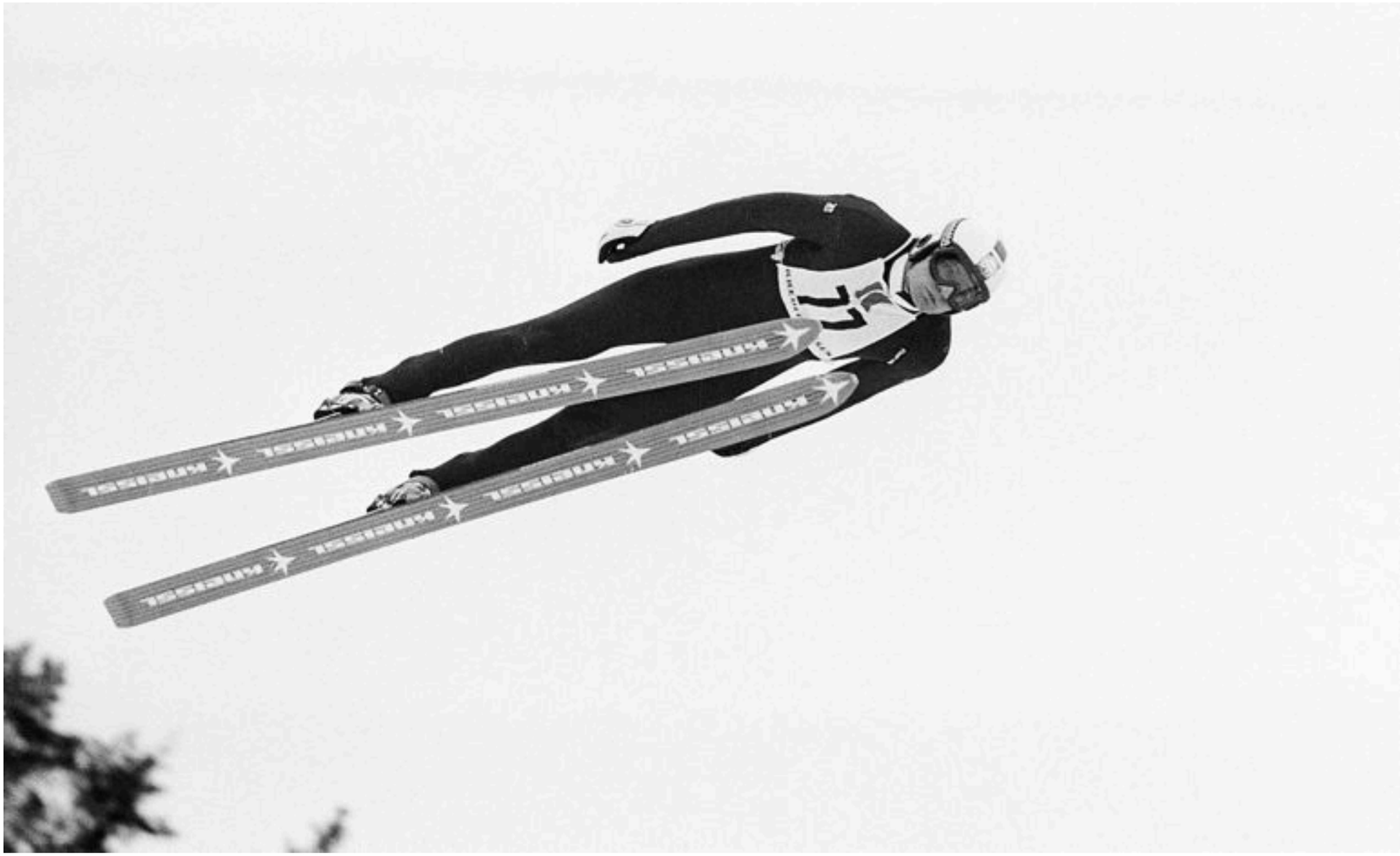
... but there was also a need for brutally efficient languages

70's
BCPL, B, C

After frustrating experience with BCPL, Bjarne Stroustrup combined the efficiency of C with some of the elegance from Simula...

80's

C with classes, C++/CFront, ARM



C++ was improved and became standardized

90's
X3J16, C++arm, WG21, C++98, STL

Ouch... Template Metaprogramming

C++03, TR1, Boost and other external libraries

While the language itself saw some minor improvements after C++98, Boost and other external libraries acted like laboratories for experimenting with potential new C++ features. Resulting in...

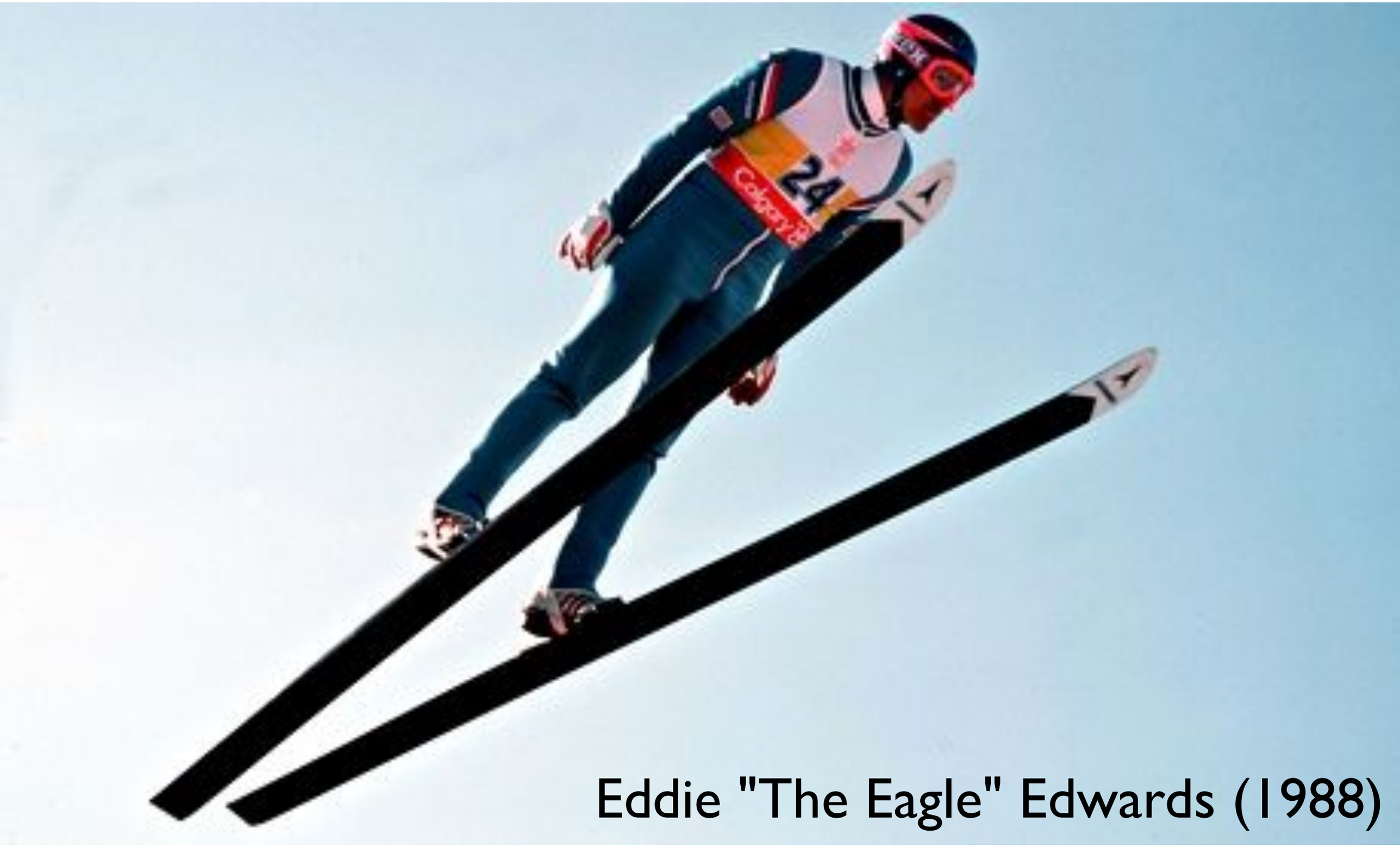
C++11/C++14

With the latest version C++ feels like a new language

The next major version is expected in 2017

The future of C++?

The success of C++ can also be meassured in that it has inspired a lot of other programming languages. Java, C# and D, just to mention a few



Eddie "The Eagle" Edwards (1988)

History of C

- CPL (1963)
- BCPL (1966)
- B (1969)
- Unix (1969-1973)
- Early C (1972)
- K&R C (1978)
- X3J11 established (1983)
- ANSI X3.159-1989 (C89 / ANSI C)
- ISO/IEC 9899:1990 (C90, same as C89)
- WG14 first meeting in 1994
- ISO/IEC 9899/AMD1:1995 (C95, minor update)
- ISO/IEC 9899:1999 (C99)
- ISO/IEC 9899:2011 (C11, current version)

History of C++

- PhD, Simula, BCPL (Cambridge)
- C with Classes (Cpre, 1979)
- First external paper (1981)
- C++ named (1983)
- CFront 1.0 (1985)
- TC++PL, Ed1 (1985)
- ANSI X3J16 meeting (1989)
- The Annotated C++ Reference Manual (1990)
- First WG21 meeting (1991)
- The Design and Evolution of C++ (1994)
- ISO/IEC 14882:1998 (C++98)
- ISO/IEC 14882:2003 (C++03)
- ISO/IEC TR 19768:2007 (C++TR1)
- ISO/IEC 14882:2011 (C++11)
- soon ISO/IEC 14882:2014 (C++14)

Modern C++ by Example

unless specified otherwise, all these code snippets should compile cleanly with a modern C++ compiler

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (std::vector<int>::const_iterator it = log.cbegin();
         it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

Consider this small toy program...

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (std::vector<int>::const_iterator it = log.cbegin();
         it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

Consider this small toy program...

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (std::vector<int>::const_iterator it = log.cbegin();
         it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

```
$ g++-4.9 -std=c++1y -Wall -Wextra -pedantic -Werror foo.cpp && ./a.out
20
24
37
42
23
45
37
$
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (std::vector<int>::const_iterator it = log.cbegin();
         it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (std::vector<int>::const_iterator it = log.cbegin();
         it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

This shows a "traditional" way of looping through a collection of objects.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (std::vector<int>::const_iterator it = log.cbegin();
         it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (std::vector<int>::const_iterator it = log.cbegin();
         it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

But why do we have to write all this stuff? In this case, wouldn't it be nice if the compiler could just figure out which type we need to store the return value from `log.cbegin()`?

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (std::vector<int>::const_iterator it = log.cbegin();
         it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (std::vector<int>::const_iterator it = log.cbegin();
         it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (decltype(log.cbegin()) it = log.cbegin();
         it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (decltype(log.cbegin()) it = log.cbegin();
         it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

`decltype` gives us type deduction in C++. Or even better...

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (auto it = log.cbegin();
         it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (auto it = log.cbegin();
         it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

We can just use the new meaning of the keyword `auto`

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (auto it = log.cbegin();
         it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (auto it = log.cbegin();
         it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (auto it = log.cbegin(); it != log.cend(); ++it)

        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (auto it = log.cbegin(); it != log.cend(); ++it)

        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (auto it = log.cbegin(); it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (auto it = log.cbegin(); it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (auto it = log.cbegin(); it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (auto it = log.cbegin(); it != log.cend(); ++it)
        transmit_item(*it);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

Looping through an array like this is something C++ programmers often do. So the language now provides a new way of looping through ranges of objects.

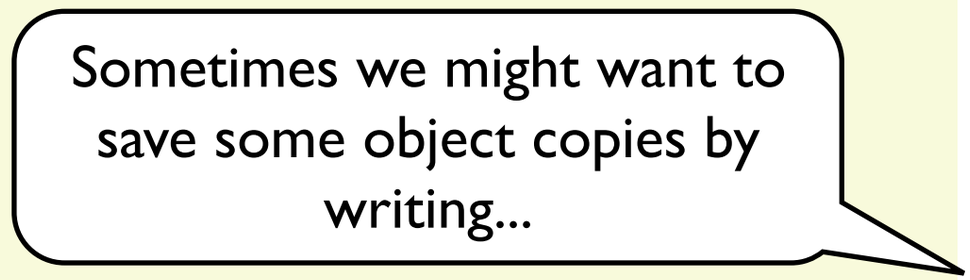
```
#include <iostream>
#include <vector>

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (auto i : log)
        transmit_item(i);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (auto i : log)
        transmit_item(i);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

Introducing:
range based for-loop.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (auto i : log)
        transmit_item(i);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (auto i : log)
        transmit_item(i);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```



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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (auto i : log)
        transmit_item(i);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

Sometimes we might want to save some object copies by writing...

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (const auto & i : log)
        transmit_item(i);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (const auto & i : log)
        transmit_item(i);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (const auto & i : log)
        transmit_item(i);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

But even for simple loops like this you will often see that **STL algorithms** are used instead.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (const auto & i : log)
        transmit_item(i);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (const auto & i : log)
        transmit_item(i);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    for (const auto & i : log)
        transmit_item(i);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    std::for_each(log.begin(), log.end(), transmit_item);

}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    std::for_each(log.begin(), log.end(), transmit_item);

}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    std::for_each(log.begin(), log.end(), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    std::for_each(log.begin(), log.end(), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    std::for_each(log.begin(), log.end(), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

You will often see this written as...

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    std::for_each(log.begin(), log.end(), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

You will often see this written as...

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

Because it now works on both containers and arrays.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

Suppose we would like to sort the array before transmitting the items...

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

Suppose we would like to sort the array before transmitting the items...

First we make a local copy of the log through a **pass-by-value**

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

Suppose we would like to sort the array before transmitting the items...

First we make a local copy of the log through a **pass-by-value**

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

But wait! What if the log has million of entries? Perhaps we should do **pass-by-reference** instead?

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

But wait! What if the log has million of entries? Perhaps we should do **pass-by-reference** instead?

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> & log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> & log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> & log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

This works. But, in this case, it would be even better if we had an option to pass the **ownership** of the log to `transmit_log` by reference so it can do whatever it wants.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> & log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

This works. But, in this case, it would be even better if we had an option to pass the **ownership** of the log to `transmit_log` by reference so it can do whatever it wants.

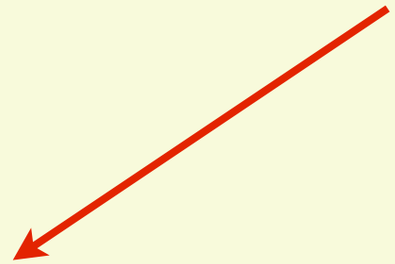
```cpp
#include <iostream>
#include <vector>
#include <algorithm>

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> & log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(log);
}
```

This works. But, in this case, it would be even better if we had an option to pass the **ownership** of the log to `transmit_log` by reference so it can do whatever it wants.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```



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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

This is an **rvalue reference**.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

This is an **rvalue reference**.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

This is an **rvalue reference**.

And here we basically say: Just take this data object, it is yours, do whatever you want with it. I promise to never refer to it again after this.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

This is an **rvalue reference**.

And here we basically say: Just take this data object, it is yours, do whatever you want with it. I promise to never refer to it again after this.

rvalue references and the corresponding **move semantics** are very important contributions to modern C++. It reduces the need to create copies of objects while still being able to use **value semantics** as a programming style (ie, avoiding the need to use pointers for everything).

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

Typical for most algorithms in the C++ library is that you can adapt them to your own needs. Let's try to change the sorting order by writing our own comparator function.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

Typical for most algorithms in the C++ library is that you can adapt them to your own needs. Let's try to change the sorting order by writing our own comparator function.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}
```

Typical for most algorithms in the C++ library is that you can adapt them to your own needs. Let's try to change the sorting order by writing our own comparator function.

```
static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static bool mycomp(int lhs, int rhs)
{
    return lhs > rhs;
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log), mycomp);
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static bool mycomp(int lhs, int rhs)
{
    return lhs > rhs;
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log), mycomp);
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static bool mycomp(int lhs, int rhs)
{
    return lhs > rhs;
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log), mycomp);
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

This is a typical way to introduce a **strategy** into an existing algorithm. (Here you could have used `std::greater` as well, but if you want something more complex you need to write it yourself.)

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static bool mycomp(int lhs, int rhs)
{
    return lhs > rhs;
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log), mycomp);
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

This is a typical way to introduce a **strategy** into an existing algorithm. (Here you could have used `std::greater` as well, but if you want something more complex you need to write it yourself.)

It depends on the context, but in this case you might want to allow the caller to pass the strategy in into `transmit_log()`

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static bool mycomp(int lhs, int rhs)
{
    return lhs > rhs;
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log), mycomp);
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

This is a typical way to introduce a **strategy** into an existing algorithm. (Here you could have used `std::greater` as well, but if you want something more complex you need to write it yourself.)

It depends on the context, but in this case you might want to allow the caller to pass the strategy in into `transmit_log()`

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static bool mycomp(int lhs, int rhs)
{
    return lhs > rhs;
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log), mycomp);
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

This is a typical way to introduce a **strategy** into an existing algorithm. (Here you could have used `std::greater` as well, but if you want something more complex you need to write it yourself.)

It depends on the context, but in this case you might want to allow the caller to pass the strategy in into `transmit_log()`

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static bool mycomp(int lhs, int rhs)
{
    return lhs > rhs;
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log), mycomp);
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

This is a typical way to introduce a **strategy** into an existing algorithm. (Here you could have used `std::greater` as well, but if you want something more complex you need to write it yourself.)

It depends on the context, but in this case you might want to allow the caller to pass the strategy in into `transmit_log()`

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static bool mycomp(int lhs, int rhs)
{
    return lhs > rhs;
}

static void transmit_log(std::vector<int> && log, bool comp(int, int))
{
    std::sort(std::begin(log), std::end(log), comp);
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), mycomp);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static bool mycomp(int lhs, int rhs)
{
    return lhs > rhs;
}

static void transmit_log(std::vector<int> && log, bool comp(int, int))
{
    std::sort(std::begin(log), std::end(log), comp);
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), mycomp);
}
```

This is an example of
parameterize from above

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static bool mycomp(int lhs, int rhs)
{
    return lhs > rhs;
}

static void transmit_log(std::vector<int> && log, bool comp(int, int))
{
    std::sort(std::begin(log), std::end(log), comp);
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), mycomp);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static bool mycomp(int lhs, int rhs)
{
    return lhs > rhs;
}

static void transmit_log(std::vector<int> && log, bool comp(int, int))
{
    std::sort(std::begin(log), std::end(log), comp);
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), mycomp);
}
```

I am now going to introduce function objects and lambdas. Let's first simplify the code, before introducing algorithms for filtering out and removing log values.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static bool mycomp(int lhs, int rhs)
{
    return lhs > rhs;
}

static void transmit_log(std::vector<int> && log, bool comp(int, int))
{
    std::sort(std::begin(log), std::end(log), comp);
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), mycomp);
}
```

I am now going to introduce function objects and lambdas. Let's first simplify the code, before introducing algorithms for filtering out and removing log values.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static bool mycomp(int lhs, int rhs)
{
    return lhs > rhs;
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static bool mycomp(int lhs, int rhs)
{
    return lhs > rhs;
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    struct filter {
        filter(int limit) : lim(limit) {}
        bool operator()(int i) { return i <= lim; };
        int lim;
    } myfilter(23);
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    struct filter {
        filter(int limit) : lim(limit) {}
        bool operator()(int i) { return i <= lim; };
        int lim;
    } myfilter(23);
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

Here we have created code to remove all log items that are 23 or below.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    struct filter {
        filter(int limit) : lim(limit) {}
        bool operator()(int i) { return i <= lim; };
        int lim;
    } myfilter(23);
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    struct filter {
        filter(int limit) : lim(limit) {}
        bool operator()(int i) { return i <= lim; };
        int lim;
    } myfilter(23);
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

Notice how we have created a "function" on the fly by overloading the call operator on an object. This is an example of a **function object**, sometimes called a **functor**.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    struct filter {
        filter(int limit) : lim(limit) {}
        bool operator()(int i) { return i <= lim; };
        int lim;
    } myfilter(23);
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    struct filter {
        filter(int limit) : lim(limit) {}
        bool operator()(int i) { return i <= lim; };
        int lim;
    } myfilter(23);
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

Suppose we want to parameterize from above again, by passing in the limit.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    struct filter {
        filter(int limit) : lim(limit) {}
        bool operator()(int i) { return i <= lim; };
        int lim;
    } myfilter(23);
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

Suppose we want to parameterize from above again, by passing in the limit.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    struct filter {
        filter(int limit) : lim(limit) {}
        bool operator()(int i) { return i <= lim; };
        int lim;
    } myfilter(23);
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

Suppose we want to parameterize from above again, by passing in the limit.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log)
{
    struct filter {
        filter(int limit) : lim(limit) {}
        bool operator()(int i) { return i <= lim; };
        int lim;
    } myfilter(23);
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log));
}
```

Suppose we want to parameterize from above again, by passing in the limit.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, int limit)
{
    struct filter {
        filter(int limit) : lim(limit) {}
        bool operator()(int i) { return i <= lim; };
        int lim;
    } myfilter(limit);
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), 23);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, int limit)
{
    struct filter {
        filter(int limit) : lim(limit) {}
        bool operator()(int i) { return i <= lim; };
        int lim;
    } myfilter(limit);
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), 23);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, int limit)
{
    struct filter {
        filter(int limit) : lim(limit) {}
        bool operator()(int i) { return i <= lim; };
        int lim;
    } myfilter(limit);
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), 23);
}
```

Such function objects are sometimes very useful. New in C++11 is a convenient syntax for creating these functions.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, int limit)
{
    struct filter {
        filter(int limit) : lim(limit) {}
        bool operator()(int i) { return i <= lim; };
        int lim;
    } myfilter(limit);
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), 23);
}
```

Such function objects are sometimes very useful. New in C++11 is a convenient syntax for creating these functions.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, int limit)
{
    struct filter {
        filter(int limit) : lim(limit) {}
        bool operator()(int i) { return i <= lim; };
        int lim;
    } myfilter(limit);
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), 23);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, int limit)
{
    struct filter {
        filter(int limit) : lim(limit) {}
        bool operator()(int i) { return i <= lim; };
        int lim;
    } myfilter(limit);
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), 23);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, int limit)
{
    auto myfilter = [limit](int i) { return i <= limit; };
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), 23);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, int limit)
{
    auto myfilter = [limit](int i) { return i <= limit; };
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), 23);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, int limit)
{
    auto myfilter = [limit](int i) { return i <= limit; };
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), 23);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, int limit)
{
    auto myfilter = [limit](int i) { return i <= limit; };
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), 23);
}
```

This is a lambda expression that creates a function object on the "fly". We are **capturing** the value of the variable `limit` and using it to initialize the function object.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, int limit)
{
    auto myfilter = [limit](int i) { return i <= limit; };
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), 23);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, int limit)
{
    auto myfilter = [limit](int i) { return i <= limit; };
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), 23);
}
```

You can of course also pass function objects around as any other objects.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, int limit)
{
    auto myfilter = [limit](int i) { return i <= limit; };
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), 23);
}
```

You can of course also pass function objects around as any other objects.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, int limit)
{
    std::function<bool (int)> myfilter = [limit](int i) { return i <= limit; };
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), 23);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, int limit)
{
    std::function<bool (int)> myfilter = [limit](int i) { return i <= limit; };
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), 23);
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, std::function<bool (int)> myfilter)
{

    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, std::function<bool (int)> myfilter)
{

    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, std::function<bool (int)> myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, std::function<bool (int)> myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, std::function<bool (int)> myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, std::function<bool (int)> myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

Basically anything that can be called with an `int` and returning a `bool` is OK. We can generalize the code with a template.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static void transmit_log(std::vector<int> && log, std::function<bool (int)> myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

Basically anything that can be called with an `int` and returning a `bool` is OK. We can generalize the code with a template.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Filt>
static void transmit_log(std::vector<int> && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Filt>
static void transmit_log(std::vector<int> && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Filt>
static void transmit_log(std::vector<int> && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

... and the same is true for the log. Anything that we can iterate over, and that contains some items that we can transmit should be fine.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Filt>
static void transmit_log(std::vector<int> && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

... and the same is true for the log. Anything that we can iterate over, and that contains some items that we can transmit should be fine.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

... and the same is true for the log. Anything that we can iterate over, and that contains some items that we can transmit should be fine.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

And while we are at it, let's generalize the code for `transmit_item` as well

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

And while we are at it, let's generalize the code for `transmit_item` as well

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log), transmit_item<int>);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),transmit_item<int>);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<int>);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<int>);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

`transmit_log` and `transmit_item` are now **type independent code**. This is a fine example of **generic programming**. Notice how we can change both the type of the log items and the container and it should still work (given some restrictions)

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

`transmit_log` and `transmit_item` are now **type independent code**. This is a fine example of **generic programming**. Notice how we can change both the type of the log items and the container and it should still work (given some restrictions)

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

`transmit_log` and `transmit_item` are now **type independent code**. This is a fine example of **generic programming**. Notice how we can change both the type of the log items and the container and it should still work (given some restrictions)

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](int i) { return i <= 23; });
}
```

`transmit_log` and `transmit_item` are now **type independent code**. This is a fine example of **generic programming**. Notice how we can change both the type of the log items and the container and it should still work (given some restrictions)

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

```
#include <iostream>
#include <deque>
#include <algorithm>
#include <functional>

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::deque<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

It would be nice to specify exactly what expectations we have to the types and objects that are passed into our generic code. A "poor man" solution is to use **type traits** and `static_assert`.

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

It would be nice to specify exactly what expectations we have to the types and objects that are passed into our generic code. A "poor man" solution is to use **type traits** and `static_assert`.

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

template <typename T>
static void transmit_item(T i)
{
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

It would be nice to specify exactly what expectations we have to the types and objects that are passed into our generic code. A "poor man" solution is to use **type traits** and `static_assert`.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <type_traits>

template <typename T>
static void transmit_item(T i)
{
    static_assert(std::is_integral<T>::value, "integral type expected");
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

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

template <typename T>
static void transmit_item(T i)
{
    static_assert(std::is_integral<T>::value, "integral type expected");
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

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

template <typename T>
static void transmit_item(T i)
{
    static_assert(std::is_integral<T>::value, "integral type expected");
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

Here we will get an understandable compile error if the type of the log items are not of integral type. However, you can, with some work, define your own traits and constraints. Eg, something like this:

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

template <typename T>
static void transmit_item(T i)
{
    static_assert(std::is_integral<T>::value, "integral type expected");
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

Here we will get an understandable compile error if the type of the log items are not of integral type. However, you can, with some work, define your own traits and constraints. Eg, something like this:

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

template <typename T>
static void transmit_item(T i)
{
    static_assert(std::is_integral<T>::value, "integral type expected");
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

Here we will get an understandable compile error if the type of the log items are not of integral type. However, you can, with some work, define your own traits and constraints. Eg, something like this:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <type_traits>
#include "mystuff"

template <typename T>
static void transmit_item(T i)
{
    static_assert(my::is_transmittable<T>::value, "transmittable type expected");
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

this is just an example that does not compile

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <type_traits>
#include "mystuff"

template <typename T>
static void transmit_item(T i)
{
    static_assert(my::is_transmittable<T>::value, "transmittable type expected");
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

this is just an example that does not compile

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <type_traits>
#include "mystuff"

template <typename T>
static void transmit_item(T i)
{
    static_assert(my::is_transmittable<T>::value, "transmittable type expected");
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

There are some proposals for the next versions of C++ to include better syntax for such constraints.

this is just an example that does not compile

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <type_traits>
#include "mystuff"

template <typename T>
static void transmit_item(T i)
{
    static_assert(my::is_transmittable<T>::value, "transmittable type expected");
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

There are some proposals for the next versions of C++ to include better syntax for such constraints.

this is just an example that does not compile

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <type_traits>
#include "mystuff"

template <typename T> require Transmittable<T>
static void transmit_item(T i)
{
    
    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <type_traits>
#include "mystuff"

template <typename T> require Transmittable<T>
static void transmit_item(T i)
{

    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <type_traits>
#include "mystuff"

template <typename T> require Transmittable<T>
static void transmit_item(T i)
{

    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <type_traits>
#include "mystuff"

template <Transmittable T>
static void transmit_item(T i)
{

    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <type_traits>
#include "mystuff"

template <Transmittable T>
static void transmit_item(T i)
{

    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <type_traits>
#include "mystuff"

template <Transmittable T>
static void transmit_item(T i)
{

    std::cout << i << std::endl;
    // ...
}

template <typename Log, typename Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <type_traits>
#include "mystuff"

template <Transmittable T>
static void transmit_item(T i)
{

    std::cout << i << std::endl;
    // ...
}

template <Iterable Log, UnaryFunctionPredicate Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <type_traits>
#include "mystuff"

template <Transmittable T>
static void transmit_item(T i)
{

    std::cout << i << std::endl;
    // ...
}

template <Iterable Log, UnaryFunctionPredicate Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <type_traits>
#include "mystuff"

template <Transmittable T>
static void transmit_item(T i)
{

    std::cout << i << std::endl;
    // ...
}

template <Iterable Log, UnaryFunctionPredicate Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

This proposal is a step towards something called Concepts. I am not going to explain that, so let's clean up the code so I can show one final thing.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <type_traits>
#include "mystuff"

template <Transmittable T>
static void transmit_item(T i)
{

    std::cout << i << std::endl;
    // ...
}

template <Iterable Log, UnaryFunctionPredicate Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <type_traits>
#include "mystuff"

template <Transmittable T>
static void transmit_item(T i)
{

    std::cout << i << std::endl;
    // ...
}

template <Iterable Log, UnaryFunctionPredicate Filt>
static void transmit_log(Log && log, Filt myfilter)
{
    log.erase(std::remove_if(std::begin(log), std::end(log), myfilter),
              std::end(log));
    std::sort(std::begin(log), std::end(log));
    std::for_each(std::begin(log), std::end(log),
                  transmit_item<typename Log::value_type>);
}

int main()
{
    using log_item_type = long;
    std::vector<log_item_type> log{20,24,37,42,23,45,37};
    transmit_log(std::move(log), [](log_item_type i) { return i <= 23; });
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    size_t items = transmit_log(log);
    std::cout << "# " << items << std::endl;
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    size_t items = transmit_log(log);
    std::cout << "# " << items << std::endl;
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    size_t items = transmit_log(log);
    std::cout << "# " << items << std::endl;
}
```

Transmitting the data probably takes some time, and we might want to do something else while waiting for the log to be transmitted. Let's simulate that, and show an example of how concurrency is supported in modern C++.

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    size_t items = transmit_log(log);
    std::cout << "# " << items << std::endl;
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    size_t items = transmit_log(log);
    std::cout << "# " << items << std::endl;
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    // ...
}

static size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    size_t items = transmit_log(log);
    std::cout << "# " << items << std::endl;
}
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <thread>

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    // ...
}

static size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    size_t items = transmit_log(log);
    std::cout << "# " << items << std::endl;
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    // ...
}

static size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    size_t items = transmit_log(log);
    std::cout << "# " << items << std::endl;
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    // ...
}

static size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    size_t items = transmit_log(log);
    std::cout << "# " << items << std::endl;
}
```

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

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    // ...
}

static size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    size_t items = transmit_log(log);
    std::cout << "# " << items << std::endl;
}
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <thread>
#include <future>

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    // ...
}

static size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    auto res = std::async(std::launch::async, transmit_log, log);
    size_t items = res.get();
    std::cout << "# " << items << std::endl;
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <thread>
#include <future>

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    // ...
}

static size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    auto res = std::async(std::launch::async, transmit_log, log);
    size_t items = res.get();
    std::cout << "# " << items << std::endl;
}
```

```
$ g++-4.9 -std=c++1y -Wall -Wextra -pedantic -Werror -pthread foo.cpp
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <thread>
#include <future>

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    // ...
}

static size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    auto res = std::async(std::launch::async, transmit_log, log);
    size_t items = res.get();
    std::cout << "# " << items << std::endl;
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <thread>
#include <future>

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    // ...
}

static size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    auto res = std::async(std::launch::async, transmit_log, log);
    size_t items = res.get();
    std::cout << "# " << items << std::endl;
}
```

... and now we can do some stuff between calling transmit_log until we need the result from calling that function.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <thread>
#include <future>

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    // ...
}

static size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    auto res = std::async(std::launch::async, transmit_log, log);
    size_t items = res.get();
    std::cout << "# " << items << std::endl;
}
```

... and now we can do some stuff between calling transmit_log until we need the result from calling that function.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <thread>
#include <future>

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    // ...
}

static size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    auto res = std::async(std::launch::async, transmit_log, log);
    for (int i=0; i<5; i++) {
        std::this_thread::sleep_for(std::chrono::milliseconds(77));
        std::cout << "do something else..." << std::endl;
    }
    size_t items = res.get();
    std::cout << "# " << items << std::endl;
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <thread>
#include <future>

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    // ...
}

static size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    auto res = std::async(std::launch::async, transmit_log, log);
    for (int i=0; i<5; i++) {
        std::this_thread::sleep_for(std::chrono::milliseconds(123));
        std::cout << "do something else..." << std::endl;
    }
    size_t items = res.get();
    std::cout << "# " << items << std::endl;
}
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono>
#include <thread>
#include <future>

static void transmit_item(int i)
{
    std::cout << i << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(200));
    // ...
}

static size_t transmit_log(const std::vector<int> & log)
{
    std::for_each(std::begin(log), std::end(log), transmit_item);
    return log.size();
}

int main()
{
    std::vector<int> log{20,24,37,42,23,45,37};
    auto res = std::async(std::launch::async, transmit_log, log);
    for (int i=0; i<5; i++) {
        std::this_thread::sleep_for(std::chrono::milliseconds(123));
        std::cout << "do something else..." << std::endl;
    }
    size_t items = res.get();
    std::cout << "# " << items << std::endl;
}
```

```
20
do something else...
24
do something else...
do something else...
37
do something else...
42
do something else...
23
45
37
# 7
```

Modern C++
- move semantics (rvalue references, value semantics)
- type deduction (decltype, auto)
- better support for OOP (attributes, member initialization, delegation)
- compile time computation (templates, static_assert, constexpr)
- template metaprogramming (traits, constraints, concepts)
- robust resource management (RAII, unique, shared)
- high-order parallelism (atomic, mutex, async, promises and futures)
- functional programming (algorithms, lamdas, closures, lazy evaluation)
- misc (chrono, user-defined literals, regex, uniform initialization)

!