C++20: All the small things

Fergus Cooper

C++ On Sea 2020

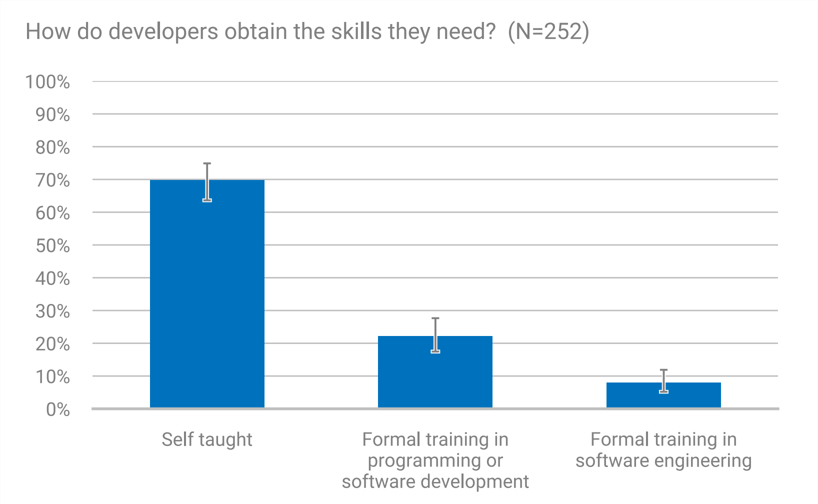
A bit about me

- Research Software Engineer at University of Oxford
- Using C++ extensively since 2014
- **C**ancer **h**eart **a**nd **s**oft **t**issue **e**nvironment (Chaste), a set of C++ libraries for
  - Cardiac electrophysiology
  - Agent-based simulations of individual cells
  - Lung physiology

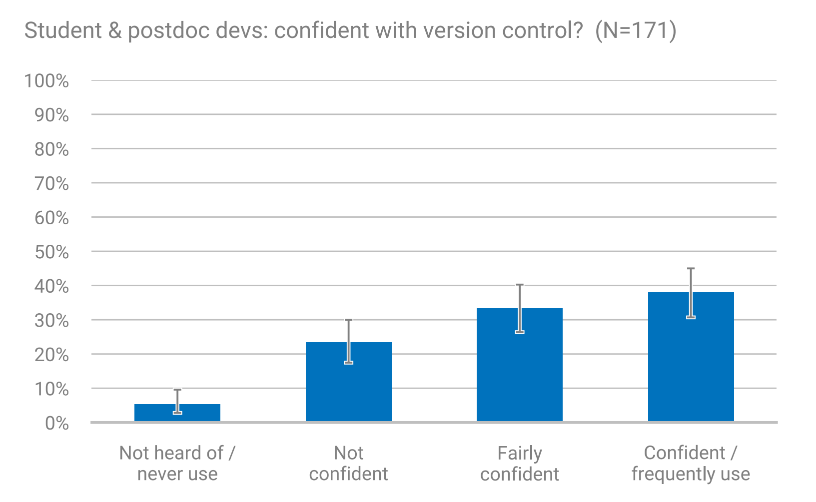
Context: C++ & software engineering in academia

- C++ is popular in academia
- 2018 survey in Oxford found C++ was 2nd after Python
  - Python
  - C++
  - MATLAB
  - R
  - C
- But, training in software engineering is **not** common

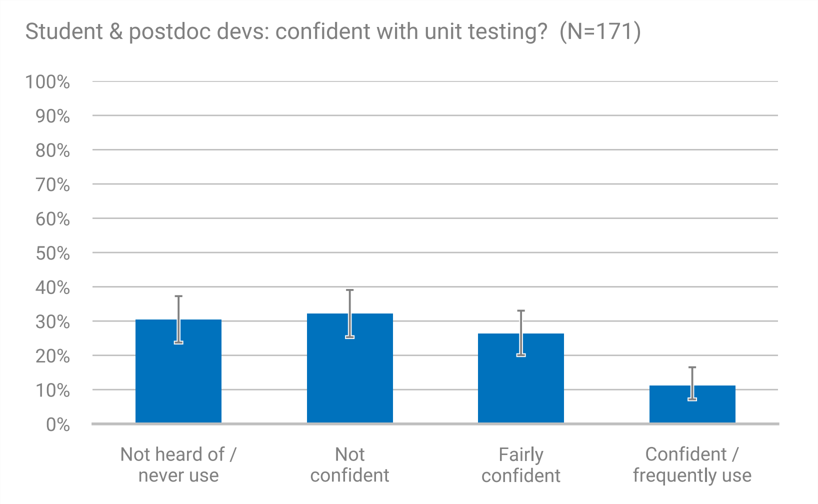
Context: C++ & software engineering in academia



Context: C++ & software engineering in academia



Context: C++ & software engineering in academia



Context: C++ & software engineering in academia

Report 9: Impact of non-pharmaceutical interventions (NPIs) to reduce COVID-19 mortality and healthcare demand

Neil M Ferguson, Daniel Laydon, Gemma Nedjati-Gilani, Natsuko Imai, Kylie Ainslie, Marc Baguelin, Sangeeta Bhatia, Adhiratha Boonyasiri, Zulma Cucunubá, Gina Cuomo-Dannenburg, Amy Dighe, Ilaria Dorigatti, Han Fu, Katy Gaythorpe, Will Green, Arran Hamlet, Wes Hinsley, Lucy C Okell, Sabine van Elsland, Hayley Thompson, Robert Verity, Erik Volz, Haowei Wang, Yuanrong Wang, Patrick GT Walker, Caroline Walters, Peter Winskill, Charles Whittaker, Christl A Donnelly, Steven Riley, Azra C Ghani.

Context: C++ & software engineering in academia

neil_ferguson
@neil_ferguson

I’m conscious that lots of people would like to see and run the pandemic simulation code we are using to model control measures against COVID-19. To explain the background - I wrote the code (thousands of lines of undocumented C) 13+ years ago to model flu pandemics...

9:13 PM · Mar 22, 2020 · Twitter for iPhone

Context: C++ & software engineering in academia

- Lots of people use C++
- Very few are experts - it's just a tool to get the job done
- Recent changes in C++ have been absolutely fantastic. They make it:
    - Easier to do the right thing
    - Harder to do the wrong thing
    - Safer by default

This talk

- There are other talks about the headline features
- This talk is about a few of my favourite little features we're getting in C++20
- Most importantly, I hope to convey why they're useful from my perspective as someone in academia

This talk

- New utilities that illustrate the progress of C++:
  - `std::midpoint`, `std::lerp`
- Better container semantics
  - `contains`, `erase` & `erase_if`, `ssize`
  - `starts_with` & `ends_with`
- New headers
  - `<source_location>` & `<numbers>`

Midpoint and linear interpolation

Midpoint and linear interpolation

- Two mathematically related functions
- `std::midpoint` in header `<numeric>`

$$\frac{a+b}{2}$$

- `std::lerp` (linear interpolation) in header `<cmath>`

$$a + t(b - a)$$

Midpoint

```cpp
const int a = 2'000'000'000;
const int b = 1'000'000'000;

std::cout << "midpoint: " << (a + b) / 2 << '\n';
```

```
>> midpoint: ???
```

Midpoint

```cpp
const int a = 2'000'000'000;
const int b = 1'000'000'000;

std::cout << "midpoint: " << (a + b) / 2 << '\n';
```

```
>> midpoint: -647483648
```

Midpoint

- `a + b` might be too large to represent as an `int`
  - So, `(a + b) / 2` won't do
- Ok, how about `a/2 + b/2`?

Midpoint

```cpp
const int a = 25;
const int b = 35;

std::cout << "midpoint: " << a / 2 + b / 2 << '\n';
```

```
>> midpoint: ???
```

Midpoint

```cpp
const int a = 25;
const int b = 35;

std::cout << "midpoint: " << a / 2 + b / 2 << '\n';
```

```
>> midpoint: 29
```

Midpoint

`a` and `b` might both round down

- So, `a/2 + b/2` won't do

Ok, how about `a + (b-a)/2`?

Midpoint

```cpp
const int a = -1'000'000'000;
const int b = 2'000'000'000;

std::cout << "midpoint: " << a + (b - a) / 2 << '\n';
```

```
>> midpoint: ???
```

Midpoint

```cpp
const int a = -1'000'000'000;
const int b = 2'000'000'000;

std::cout << "midpoint: " << a + (b - a) / 2 << '\n';
```

```
>> midpoint: -1647483648
```

Midpoint

Back to where we started: possibility of overflow. So how can it safely be done?

```
int midpoint(const int a, const int b) {
  int direction = 1;
  unsigned lo = a;
  unsigned hi = b;
  if (a > b) {
    direction = -1;
    lo = b;
    hi = a;
  }
  return a + direction * int(unsigned(hi - lo) / 2);
}
```

(Implementation based on libstdc++ 9)

Midpoint

And it's different for floating point types:

```
float midpoint(const float a, const float b) {
  float lo = std::numeric_limits<float>::min() * 2;
  float hi = std::numeric_limits<float>::max() / 2;
  float abs_a = std::fabs(a);
  float abs_b = std::fabs(b);
  if (abs_a <= hi && abs_b <= hi) [[likely]]
    return (a + b) / 2;
  if (abs_a < lo)
    return a + b / 2;
  if (abs_b < lo)
    return a / 2 + b;
  return a / 2 + b / 2;
}
```

(Implementation based on libstdc++ 9)

Midpoint

- Uses, often as a building block:
  - Anywhere you need the mean of two numbers: median?

```
float median(std::vector<float> &v) {

  auto half_way = v.size() / 2;
  std::nth_element(v.begin(), v.begin() + half_way,
                   v.end());

  if (v.size() % 2 == 1) {
    return v.at(half_way);
  } else {
    std::nth_element(v.begin(), v.begin() + half_way - 1,
                     v.begin() + half_way);
    return std::midpoint(v.at(half_way),
                         v.at(half_way - 1));
  }
```

Linear interpolation

- For floating point a, b and t, return $a + t(b - a)$
  - Interpolation if $t \in [0, 1]$, extrapolation otherwise
- Desirable properties:
  - `lerp(a,b,0) == a`
  - `lerp(a,b,1) == b`
  - monotonicity in `t`
  - if a and b are finite and $t \in [0, 1]$, then `lerp(a,b,t)` is finite

Linear interpolation

- The problem, again, is obvious implementations aren't quite right:
- $a + t(b - a)$
  - could overflow
  - when `t==1`, not guaranteed to return `b`
- $(1 - t)a + tb$
  - not guaranteed to be monotonic (unless $ab \leq 0$)

Linear interpolation

```
float lerp(float a, float b, float t) {
  if (a <= 0 && b >= 0 || a >= 0 && b <= 0)
    return t * b + (1 - t) * a;
  if (t == 1)
    return b;
  const float x = a + t * (b - a);
  return t > 1 == b > a ? std::max(b, x) : std::min(b, x);
}
```

(Implementation based on libstdc++ 9)

Linear interpolation

Uses:

- computer graphics
- colour maps
- evenly-spacing points around a polygon
- building block for other algorithms
    - bilinear interpolation

Linear interpolation

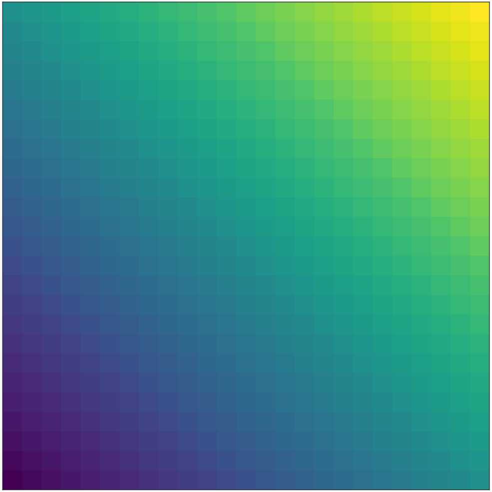
Bilinear interpolation can be implemented with three `lerp`s:

```
float bilinear(
        float x, float y,
        float x1y1, float x1y2, float x2y1, float x2y2) {

  float interp1 = std::lerp(x1y1, x2y1, x);
  float interp2 = std::lerp(x1y2, x2y2, x);

  return std::lerp(interp1, interp2, y);
}
```

Linear interpolation



Midpoint and linear interpolation: remarks

- Two excellent examples of simple functions that are non-trivial to implement correctly
- Speculation: a tiny proportion of C++ users have a CS degree
- Common building blocks like `midpoint` and `lerp` are excellent additions to the standard library:
  - No time wasted re-inventing standard tools
  - No chance of accidentally getting it wrong
  - Common enough(?) to justify inclusion

Short pause for questions

Better container semantics

Associative containers `contain`

A new member function for `map`, `multimap`, `set`, `multiset` (& unordered versions)

Checking if an element exists is very unintuitive for beginners:

```
std::set<char> s = {'a', 'b', 'c', 'd'};
if(s.find('c') != s.end()) {/* */}
```

From C++20 this is simplified with a small quality-of-life improvement:

```
std::set<char> s = {'a', 'b', 'c', 'd'};
if(s.contains('c')) {/* */}
```

Reduces consistency with other (non-associative) containers?

Consistent container erasure

- Speaking of idioms that beginners find difficult: erasing elements from containers

- Take `std::vector<>::erase`:
  - it takes one (or two) iterators and erases one (or a range of) elements: so you first have to find the elements you're looking for
  - there's an algorithm for finding things! `remove` (or `remove_if`)
  - so we've found things we want to erase with `remove`, which ~~re~~**moves** everything ethem to the back of the vector, then we erase the removed elements
  - simple?

```
std::vector<char> v = {'a', 'b', 'c', 'd', 'e'};
v.erase(std::remove(v.begin(), v.end(), 'c'), v.end());
```

- C++20 adds free functions `erase` and `erase_if` that "do what you expect". Let's see them in action:

Consistent container erasure

```
auto pred = [](char cmp){return cmp > 'c';};

std::set<char> set = {'a', 'b', 'c', 'd', 'e'};
std::cout << set.size() << '\n';                    // 5
std::cout << std::erase_if(set, pred) << '\n';  // 2
std::cout << set.size() << '\n';                    // 3

std::vector<char> vec = {'a', 'b', 'c', 'd', 'e'};
std::cout << std::erase_if(vec, pred) << '\n';

std::string str = "abcde";
std::cout << std::erase_if(str, pred) << '\n';
```

(Nearly) consistent container erasure

- This is great... ish
- It doesn't seem quite "consistent"

We have gained `std::erase` overloads for:

- `basic_string`, `deque`, `vector`, `forward_list`, `list`

And `std::erase_if` overloads for the above, plus:

- `map`, `multimap`, `set`, `multiset` (and their unordered counterparts)

We have to now remember which containers only have the member `erase`.
Hmm.

Signed size

Containers can be queried for their size, which is unsigned:

```
std::vector<int> v = {1, 2, 3, 4, 5, 6};

for (int i = 0; i < v.size(); ++i) {
  /* */
}
```

Comparison between `signed` and `unsigned`.

Signed size

This isn't necessarily a problem by itself, but other patterns are more dangerous:

```
bool has_repeated_values(std::vector<int> &container) {
  for (int i = 0; i < container.size() - 1; ++i) {
    if (container[i] == container[i + 1]) {
      return true;
    }
  }
  return false;
}
```

(Example adapted from P1227 by Jorg Brown)

My IDE did not warn me about potential problems with this code, but...

Signed size

The following will cause problems:

```
std::vector<int> empty_vec = {};
has_repeated_values(empty_vec);  // ???
```

A member `ssize()` method returning a signed integer would solve this class of problems (if used).

Signed size

Unfortunately, we only got a compromise `std::ssize()` free function:

```
std::vector<int> v = {1, 2, 3, 4, 5, 6};
for (int i = 0; i < std::ssize(v); ++i) {/**/}
```

Better still to use range-for or stl algorithms where possible:

```
bool has_repeated_values(std::vector<int> &v) {
  return std::adjacent_find(v.begin(), v.end()) != v.end();
}
```

Signed size

- But when not possible, `std::ssize()` makes it easier to "do the right thing"
- A missed opportunity not having member functions?
  - most C++ programmers don't know their standard library inside out
  - likely to pick from the list of member functions that their IDE gives them
- We now have `size` member and free functions, but only `ssize` free functions: consistency?
- Better tooling to suggest using `std::ssize()` when `std::size()` or `.size()` are used and would compare types of different signedness?

String utilities

`starts_with` and `ends_with`: indispensable member functions!

Pre-C++20 checking whether a string ends with another string is not beginner friendly:

```
bool ends_with(std::string &orig, std::string &ending) {
  if (orig.length() >= ending.length()) {
    return (orig.compare(orig.length() - ending.length(),
                         ending.length(), ending) == 0);
  } else {
    return false;
  }
}
```

- Need to know how `compare` works
- Need to get remember the length-check

String utilities

We now get two new member functions: `starts_with` and `ends_with`, which makes this kind of code possible:

```
std::vector<fs::path> data_files;
for (auto &p :
        fs::recursive_directory_iterator("data_dir")) {
  if (p.path().string().ends_with(".dat")) {
    data_files.emplace_back(p.path());
  }
}
```

- Intuitive, easy to find, common to want

Better container semantics: remarks

- Many small improvements to containers that make life:
    - safer for non-experts
    - easier for all
    - a little less frustrating
- Removes several idioms that must be taught
- Reduces stack overflow's carbon footprint?
- Seem(?) to have stopped just short of consistency and simplicity
    - Is there a good reason not have have `erase` for `set`?
    - Is there a good reason not have have member `ssize()` methods?

Short pause for questions

New headers

<source_location>

Access the caller's file name, line number and column, without macros.

```
void log(std::string_view message,
         std::source_location location =
                 std::source_location::current()) {
  std::cout << location.file_name() << ':'
            << location.line() << ':'
            << location.column() << ' '
            << message << '\n';
}

int main() {
  log("message!");  // path/to/main.cpp:12:0 message!
}
```

Not yet implemented, except in GCC's `std::experimental`.

<source_location>

A real-world example from Chaste:

```
#define MARK std::cout << __FILE__\
<< " at line " << __LINE__ << std::endl;
}
```

```
void mark(std::source_location location =
                  std::source_location::current()) {
  std::cout << location.file_name() << " at line "
            << location.line() << std::endl;
}
```

One more macro that can be removed! (One day.)

Unfortunately we still can't do anything about the following kind of macro:

```
#define PRINT_VARIABLE(var) std::cout << #var\
" = " << var << std::endl;
```

We will have to wait for reflection...

Mathematical constants: <numbers>

The C++ standard has a lot of maths in it:

- `exp`, `log`, `pow`, `sqrt`, ...
- `lerp`
- `comp_ellint_2`, `cyl_bessel_k`, `sph_neumann`, ...

But until C++20 there was no definition of mathematical constants such as π and e.

- `<math.h>` tends to define macros
- Microsoft defines macros if you `#define _USE_MATH_DEFINES` before you `#include <cmath>`

```
# define M_PI    3.14159265358979323846
# define M_E   2.7182818284590452354
```

```
# define M_PIl  3.141592653589793238462643383279502884L
```

Mathematical constants: <numbers>

But we don't like macros, so how can we replace these?

We could expose constants:

```
constexpr double pi = 3.14159265358979323846;
constexpr double e =  2.7182818284590452354;
```

But then we're still stuck with the problem of not having `float` or `long double` versions...

Mathematical constants: <numbers>

C++14 introduced ‘variable templates’ which lets us define templated constants:

```
template<typename FloatingType>
constexpr FloatingType pi =
        static_cast<FloatingType>(3.1415926535897932385L);
```

```
const float pi_f = pi<float>;
const double pi_d = pi<double>;
const long double pi_l = pi<long double>;
```

Mathematical constants: <numbers>

In the end, we got both:

```
const auto pi_f = std::numbers::pi_v<float>;
const auto pi_d = std::numbers::pi_v<double>;
const auto pi_l = std::numbers::pi_v<long double>;

const double pi = std::numbers::pi;
```

$e \quad \log_2(e) \quad \log_{10}(e) \quad \log_e(2) \quad \log_e(10)$

$\pi \quad \frac{1}{\pi} \quad \frac{1}{\sqrt{\pi}} \quad \sqrt{2} \quad \sqrt{3} \quad \frac{1}{\sqrt{3}} \quad \gamma \quad \phi$

Mathematical constants: <numbers>

And finally, unlike the Indiana General Assembly, the C++ standard does not attempt to legislate the value of any of these constants.

Indiana Pi Bill

From Wikipedia, the free encyclopedia

The **Indiana Pi Bill** is the popular name for bill #246 of the 1897 sitting of the Indiana General Assembly, one of the most notorious attempts to establish mathematical truth by legislative fiat.

Instead:

26.9.2 Mathematical constants **[math.constants]**

1 The library-defined partial specializations of mathematical constant variable templates are initialized with the nearest representable values of e, $\log_2 e$, $\log_{10} e$, π, $\frac{1}{\pi}$, $\frac{1}{\sqrt{\pi}}$, $\ln 2$, $\ln 10$, $\sqrt{2}$, $\sqrt{3}$, $\frac{1}{\sqrt{3}}$, the Euler-Mascheroni γ constant, and the golden ratio ϕ constant $\frac{1+\sqrt{5}}{2}$, respectively.

Thanks for listening. Any questions?