accu
2023

ARE THE OLD WAYS SOMETIMES THE BEST?

ROGER ORR

Are the old ways sometimes the best?

Roger Orr

OR/2 Limited

Comparing the 'classic C++' and 'modern' ways to solve various programming tasks

What are some of the trade-offs?

– April 2023 –

Are the old ways sometimes the best?

- Programming languages change over time; sometimes these changes give *new* ways of doing old things.
- As the dust settles on C++23 I was reflecting on some of the lessons we might learn from places in C++ where this has occurred since its inception.
    - What might guide us in choosing between idioms?
    - Any lessons for code we ourselves produce for others to consume?

The `for` loop

- We begin with a simple task: calculate the produce of a vector of integers. Here's how it might have looked in C++98...

```
#include <vector>


int product(const std::vector<int>& items) {
  int result = 1;
  for (int i = 0; i < items.size(); i++) {
    result *= items[i];
  }
  return result;
}
```

The `for` loop

- Is this code **understandable** ?
  Is this code **right** ?

```
#include <vector>


int product(const std::vector<int>& items) {
  int result = 1;
  for (int i = 0; i < items.size(); i++) {
    result *= items[i];
  }
  return result;
}
```

The `for` loop

- This code produces the *same assembler output** as the for loop...

```
int product_raw(const std::vector<int>& items) {
  int result = 1;
  int i = 0;
  goto end;
  loop:;
    result *= items[i];
    ++i;
  end:
    if (i < items.size()) goto loop;
  return result;
}
```

- **Would anyone prefer this code to the original?**

(* With a couple of different compilers, but YMMV)

The `for` loop

- There are some small changes that a “modern” code review might suggest to this code

```
#include <vector>

int product(const std::vector<int>& items) {
  int result{1};
  for (std::size_t idx{}; idx != items.size(); ++idx) {
    result *= items[idx];
  }
  return result;
}
```

The `for` loop

- The “Almost Always Auto” school of thought might suggest other changes:

```
#include <vector>

int product(const std::vector<int>& items) {
  auto result{1};
  for (auto idx{0uz}; idx != items.size(); ++idx) {
    result *= items[idx];
  }
  return result;
}
```

(Using P0330R8 “Literal Suffix for (signed) size_t” from C++23, in gcc, clang, & edg)

The `for` loop

- Hopefully no-one would suggest *this* change (unless for some reason you have to iterate in reverse):

```
int product(const std::vector<int>& items) {
  auto result{1};
  for (auto idx{items.size()}; idx--; ) {
    result *= items[idx];
  }
  return result;
}
```

(`idx--` or `--idx`; and are you sure...?)

The `for` loop

- Or *this* one:

```
int product(const std::vector<int>& items) {
  if (items.empty()) return 1;
  auto result{items[0]};
  for (std::size_t idx{1}; idx != items.size(); ++idx) {
    result *= items[idx];
  }
  return result;
}
```

The `for` loop

- Some would prefer an iterator solution. Here's a C++98 style:

```
int product(const std::vector<int>& items) {
  int result = 1;
  for (std::vector<int>::const_iterator it = items.begin();
       it != items.end(); ++it) {
    result *= *it;
  }
  return result;
}
```

The `for` loop

- Here is what a more modern writer might use:

```
int product(const std::vector<int>& items) {
  auto result = 1;
  for (auto it = items.cbegin(); it != items.cend(); ++it) {
    result *= *it;
  }
  return result;
}
(cbegin/cend added in C++11)
```

The `for` loop

- An iterator solution can be **generalised**:

```
template <typename Coll>
int product(const Coll& items) {
  int result = 1;
  for (auto it = items.cbegin(); it != items.cend(); ++it) {
    result *= *it;
  }
  return result;
}
```

The `for` loop

- Moving away from “raw” loops, there's also this language solution:

```
int product(const std::vector<int>& items) {
  int result = 1;
  for (auto item : items) {
    result *= item;
  }
  return result;
}
```

The `for` loop

- This further simplification didn't achieve consensus, however:

```
int product(const std::vector<int>& items) {
  int result = 1;
  for (item : items) {
    result *= item;
  }
  return result;
}
```

(N3994, rejected in plenary)

The `for` loop

- Sean Parent has a phrase “No raw loops”. He says*
- Use an existing algorithm
  - Prefer standard algorithms if available
- Implement a known algorithm as a general function
  - Contribute it to a library
  - Preferably open source
- Invent a new algorithm
  - Write a paper
  - Give talks
  - Become famous!

* https://sean-parent.stlab.cc/presentations/2013-09-11-cpp-seasoning/cpp-seasoning.pdf

The `for` loop

- This is a simple way to use an algorithm:

```
#include <algorithm>


int product(const std::vector<int>& items) {
  int result = 1;
  std::for_each(items.begin(), items.end(),
    [&](int item) { result *= item; } );
  return result;
}
```

The `for` loop

- Or a *different* algorithm:

```
#include <functional>
#include <numeric>

int product(const std::vector<int>& items) {
  return std::accumulate(items.begin(), items.end(), 1,
    std::multiplies<>());
}
```

(or, pre C++14 (N3421 “Making Operator Functors greater<>”), `std::multiplies<int>`)

The `for` loop

- Or *another* different algorithm:

```
#include <functional>
#include <numeric>

int product(const std::vector<int>& items) {
  return std::ranges::fold_left(items, 1, std::multiplies<>());
}
```

(Using P2322R6 “ranges::fold” in C++23, implemented in MSVC)

The `for` loop - summary

- The simple `for` loop can be re-written in a lot of different ways.
- Which way best expresses intent … to *your* code's audience
- Non-idiomatic loop constructs are harder to reason about
- Hiding the loop completely can avoid having to think about it

Constraining templates

- Consider a generalised function from the earlier for loop example:

```
#include <functional>

#include <numeric>


template <template <typename...> typename Coll, typename U>

Coll<U>::value_type product(const Coll<U>& items) {

  return std::accumulate(items.begin(), items.end(), 1,
    std::multiplies<>());

}
```

- Prior to C++20 we needed '`typename`' before `Coll<U>::value_type` – fixed with P0634 “Down with `typename`!”
- Prior to C++17 we needed class for template template parameters (N4051)

Constraining templates

- Consider a generalised function from the earlier for loop example:

```
#include <functional>

#include <numeric>


template <template <typename...> typename Coll, typename U>

auto product(const Coll<U>& items) {

  return std::accumulate(items.begin(), items.end(), 1,
    std::multiplies<>());

}
```

- **Or we may prefer a deduced return type**

Constraining templates

- However, what about these usages?

```
void good(std::vector<int>& ints) {
  std::cout << product(ints);
}

void bad(std::vector<std::string>& strings) {
  std::cout << product(strings); // <-- error
}
```

Instantiation of the 'product' function for `U = std::string` produces errors*, and in general we may wish to provide another overload to use

(*which I am sparing you)

Constraining templates

- The original way used **`std::enable_if`** (since C++11):

```
#include <type_traits>
template <template <typename> typename T, typename U,
typename = std::enable_if_t<std::is_arithmetic_v<U>>>
auto product(const T<U>& items) {
  return std::accumulate(items.begin(), items.end(), 1,
    std::multiplies<>());
}
```

Now `product()` will no longer instantiate for U `= std::string` and the error message no longer refers only to the implementation item that fails to compile.
We could also **overload** with a different constraint.

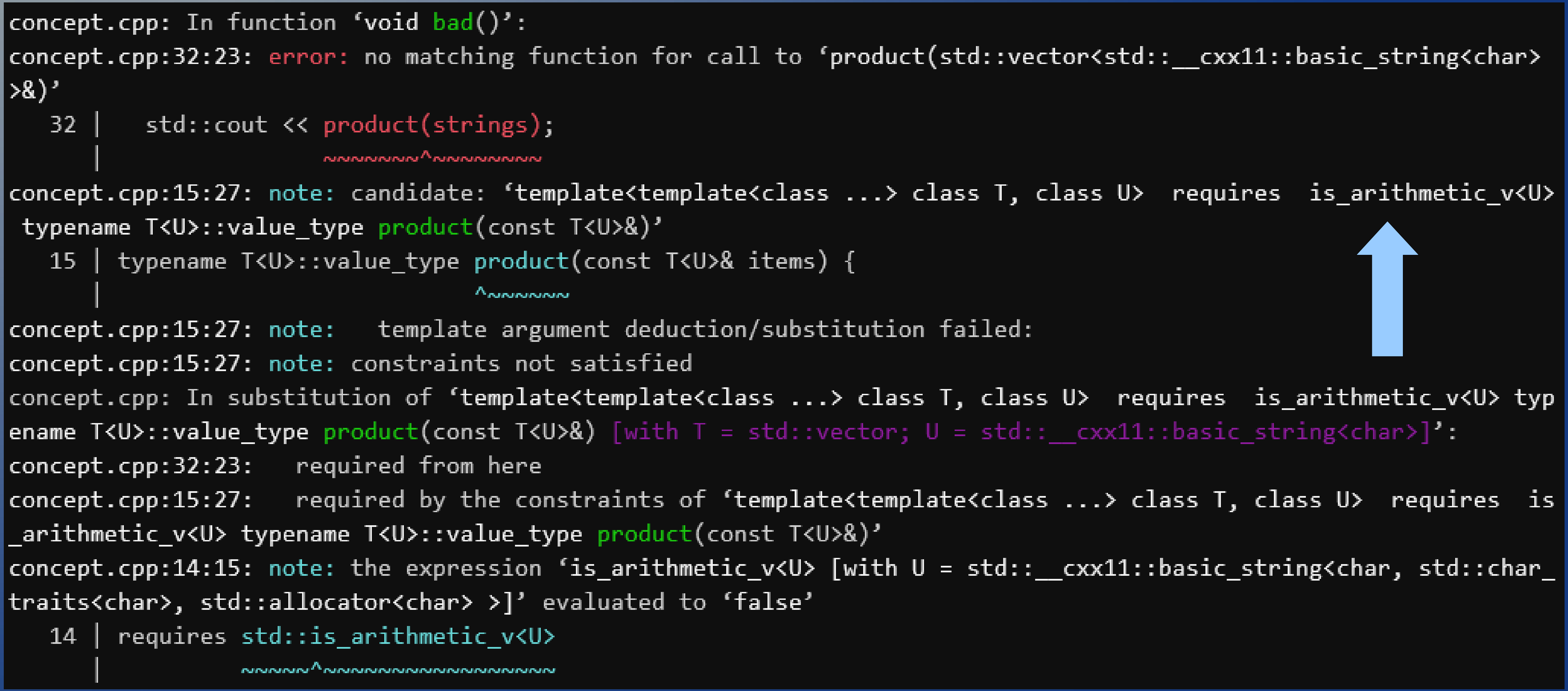
Constraining templates

- With concepts (since C++20) we can use a **requires** clause:

```
#include <type_traits>
template <template <typename...> typename T, typename U>
requires std::is_arithmetic_v<U>
auto product(const T<U>& items) {
  return std::accumulate(items.begin(), items.end(), 1,
    std::multiplies<>());
}
```

This is very similar to the `enable_if` example.

Constraining templates



```
concept.cpp: In function ‘void bad()’:
concept.cpp:32:23: error: no matching function for call to ‘product(std::vector<std::__cxx11::basic_string<char>
>&)’
   32 |   std::cout << product(strings);
      |                ~~~~~~~^~~~~~~~~
concept.cpp:15:27: note: candidate: ‘template<template<class ...> class T, class U>  requires  is_arithmetic_v<U>
 typename T<U>::value_type product(const T<U>&)’
   15 | typename T<U>::value_type product(const T<U>& items) {
      |                           ^~~~~~~
concept.cpp:15:27: note:   template argument deduction/substitution failed:
concept.cpp:15:27: note: constraints not satisfied
concept.cpp: In substitution of ‘template<template<class ...> class T, class U>  requires  is_arithmetic_v<U> typ
ename T<U>::value_type product(const T<U>&) [with T = std::vector; U = std::__cxx11::basic_string<char>]’:
concept.cpp:32:23:   required from here
concept.cpp:15:27:   required by the constraints of ‘template<template<class ...> class T, class U>  requires  is
_arithmetic_v<U> typename T<U>::value_type product(const T<U>&)’
concept.cpp:14:15: note: the expression ‘is_arithmetic_v<U> [with U = std::__cxx11::basic_string<char, std::char_
traits<char>, std::allocator<char> >]’ evaluated to ‘false’
   14 | requires std::is_arithmetic_v<U>
      |          ~~~~~^~~~~~~~~~~~~~~~~
```

Constraining templates

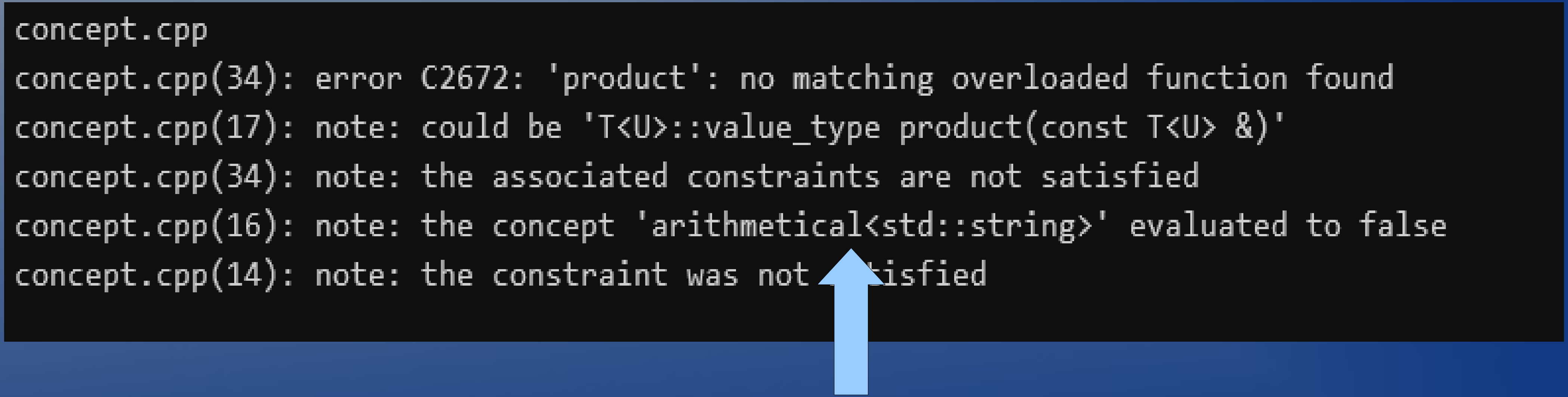
- Or we can use a **concept** in the parameter list (we must define a concept in *this* case as the standard doesn't):

```
#include <type_traits>
template <typename C> concept arithmetical =
  std::is_arithmetic_v<C>;

template <template <typename...> typename T, arithmetical U>
auto product(const T<U>& items) {
  return std::accumulate(items.begin(), items.end(), 1,
    std::multiplies<>());
}
```

Constraining templates

- This gave me the clearest error message:



```
concept.cpp
concept.cpp(34): error C2672: 'product': no matching overloaded function found
concept.cpp(17): note: could be 'T<U>::value_type product(const T<U> &)'
concept.cpp(34): note: the associated constraints are not satisfied
concept.cpp(16): note: the concept 'arithmetical<std::string>' evaluated to false
concept.cpp(14): note: the constraint was not satisfied
```

Constraining templates - summary

- Some advantages of using concepts over `enable_if` are:
  - Using a **language** feature **reduces** the wording
  - **better interaction** with overload resolution, as the constraints are considered when ordering candidates
  - may have better compilation times than enable_if
- A named concept is usually clearer than a requires expression (and works better with overload resolution)
- In this case I don't personally know of a good reason to use the old way (enable_if)

Streaming messages to/from memory

- The C++ “hello world” program used for years demonstrates the streaming idiom:

```
#include <iostream>
int main() {
  std::cout << "Hello world" << std::endl;
}
```

- The streaming paradigm is also usable in-memory too.
- Possible examples:
  - Logging (often hidden inside a macro)
  - Parsing input strings obtained elsewhere

Streaming messages to/from memory

- We often want to log information about values during the execution of a program and the streaming paradigm gives us a well recognised way to do so. It is often buried inside a macro, but expands to:

```
extern "C" void log(const char *str); // for example


...
  oss << "Input " << 42 << " produced " << result
      << " at line " << __LINE__;
  log(oss.str());
...
```

Streaming messages to/from memory

- Back in the old days pre-C++98 we used **<strstream>**:

```
#include <strstream>


...
  std::ostrstream oss;
  oss << "Input " << 42 << " produced " << result
      << " at line " << __LINE__;
  log(oss.str());
...
```

Streaming messages to/from memory

- Conversely, you can use input streams to extract fields from a string; here how you might do this, again using pre-C++98:

```
#include <strstream>

void area::configure(const std::string& value) {
  std::istrstream iss(value.c_str());
  if (!(iss >> width >> height)) {
    throw std::runtime_error("Error in '" + value + "'");
  }
  ...
}
```

Streaming messages to/from memory

- Back in the old days we used <strstream> **but there were pitfalls**:

```
#include <strstream>

  std::ostrstream oss;
  oss << "Input " << 42 << " produced " << result
      << " at line " << __LINE__;
  log(oss.str()); // Whoops
  // no terminating NUL (std::ends is needed)
  // memory leaks unless you call oss.freeze(false)
...
```

Streaming messages to/from memory

- This approach was superceded in C++98 with <sstream>

```
#include <sstream>

  std::ostringstream oss;
  oss << "Input " << 42 << " produced " << result
      << " at line " << __LINE__;
  log(oss.str().c_str());
  // terminating NUL guaranteed, no memory leak
 ...
```

Streaming messages to/from memory

- This approach was superceded in C++98 with <sstream>

```
#include <sstream>

void area::configure(const std::string& value) {
  std::istringstream iss(value~~.c_str()~~); // ctor takes string
  if (!(iss >> width >> height)) {
    throw std::runtime_error("Error in '" + value + "'");
  }
  ...
}
```

Streaming messages to/from memory

- This approach was superceded in C++98 with <sstream> - but...

```
#include <sstream>


  std::ostringstream oss;
  oss << "Input " << 42 << " produced " << result
      << " at line " << __LINE__;
  log(oss.str().c_str()); // Message contents duplicated by str()
 ...
```

Streaming messages to/from memory

- This approach was superceded in C++98 with <sstream> - but they are **still** in C++23 marked as deprecated.
- Move semantics: added in **C++20** by Peter Sommerlad's P0408R7

```
std::ostringstream oss;
oss << "Input " << 42 << " produced " << result
    << " at line " << __LINE__;
log(std::move(oss).str().c_str()); // message payload moved
...
```

- So, is everyone happy now?

Streaming messages to/from memory

- You could use an **external buffer** in the “old days”

```
...
  char fixed_buffer[40];
  std::ostrstream oss(fixed_buffer, sizeof(fixed_buffer));
  oss << "Input " << 42 << " produced " << result
      << " at line " << __LINE__ << std::ends;
  log(oss.str());
 …
```

- If I change to use ostringstream it **copies the string**

Streaming messages to/from memory

- Sigh. Ok, now you can in modern C++ too, P0448R4 (also by Peter Sommerlad) adds **span** streams to C++23:

```
#include <spanstream>
...
  char fixed_buffer[40];
  std::ospanstream oss(fixed_buffer, sizeof(fixed_buffer));
  oss << "Input " << 42 << " produced " << result
      << " at line " << __LINE__ << std::ends;
  log(oss.span().data());
 ...
```

Streaming messages to/from memory

- If you have a C++ interface that takes a `std::string_view*`, then you can use it directly:

```
#include <spanstream>
...
  char fixed_buffer[40];
  std::ospanstream oss(fixed_buffer, sizeof(fixed_buffer));
  oss << "Input " << 42 << " produced " << result
      << " at line " << __LINE__ << std::ends;
  log(oss.span());
 …
```

(* or a pointer and a length)

Streaming messages to/from memory

- There is also an **input** span view, avoiding needing a string at all:

```
#include <spanstream>
...
  void configure(const std::string& value) {
  std::ispanstream iss(value);
  if (!(iss >> width >> height)) {
    throw std::runtime_error("Error in '" + value + "'");
  }
  ...
}
```

Streaming messages to/from memory - summary

- I think that `strstream` is a salutary example of an unexpectedly long lived class
- While sstream is a cleaner design the extra encapsulation reduced take-up – something to consider when trying to provide an upgrade path: what might *prevent* someone upgrading

Initializing objects

- One of the important concepts in C++ is that of the constructor: ensuring that objects are correctly initialized.
- For example, let's consider this simplified class:

```
class point {
  int x, y;
  double distance;
public:
  point(int x, int y);
  point(const std::pair<int, int>& coord);
  // accessors, etc…
};
```

Initializing objects

- One way to write the constructors could be, for instance:

```
point::point(int x, int y) :
  x(x),
  y(y),
  distance(std::sqrt(x*x + y*y)) {}


point::point(const std::pair<int, int>& coord) :
  x(coord.first),
  y(coord.second),
  distance(std::sqrt(x*x + y*y)) {}
```

There is some near duplication here – can we avoid it?

Initializing objects

- In C++98 one way was to use a helper member function:

```
point::point(int x, int y) :
  x(x),
  y(y) {
  init();
}

point::point(const std::pair<int, int>& coord) :
  x(coord.first),
  y(coord.second) {
  init();
}

void point::init() {
  distance = std::sqrt(x*x + y*y);
}
```

Initializing objects

- Forwarding constructors allows us to *chain* the calls:

```
point::point(int x, int y) :
  x(x),
  y(y),
  distance(std::sqrt(x*x + y*y)) {}

point::point(const std::pair<int, int>& coord) :
  point(coord.first, coord.second) {}
```

- The code is shorter and, I believe, expresses the intent more clearly. It is also more resilient in the face of future changes, such as the classic case of adding a field which is not initialized correctly in one of the several constructors.
- There *may* be a performance impact, esp. in unoptimised builds

Initializing objects

- Non-static member initializers also allow us to express the constraint:

```
class point {
  int x, y;
  double distance = std::sqrt(x*x + y*y);
  // … etc
```

And in the implementation file:

```
point::point(int x, int y) :
  x(x),
  y(y)~~,~~
~~ distance(std::sqrt(x*x + y*y)) ~~{}

point::point(const std::pair<int, int>& coord) :
  x(coord.first),
  y(coord.second)~~,~~
~~  distance(std::sqrt(x*x + y*y)) ~~{}
```

Initializing objects

- In this case note that the header file must now `#include <cmath>`. Or of course you could provide a private static helper method:

  ```
  class point {
    int x, y;
    double distance = calculate(x, y);
    static double calculate(int x, int y);
    // … etc
  ```

  And in the implementation file:

  ```
  double point::calculate(int x, int y) {
    return distance(sqrt(x*x + y*y));
  }
  ```

Initializing objects

Should immutable data members be const or non-const?

```
class point {
  const int x, y;
  const double distance;
public:
  point(int x, int y);
  point(const std::pair<int, int>& coord);
  // …
};
```

What does this **allow** and **disallow**?

C++ Core Guidelines C.12: Don’t make data members `const` or references

“They are not useful, and make types difficult to use by making them either uncopyable or partially uncopyable for subtle reasons.”

Initializing objects - summary

- Know the various different mechanisms for initializing member data
- Non-static data member initializers are great, but we aware of the possible downsides

Using tuple and pair

The standard library tuple and pair classes can make good vocabulary types

- “*The types that are most commonly passed through interfaces in a given codebase are what we call “vocabulary types” - these are the most common generic forms of data for any project.” - Titus Winters

```
#include <tuple>

auto create_entry() {
  int key{};
  int value{};
  // ...
  // stuff happens
  // ...
  return std::tuple<int, int>{key, value};
}
```

Using tuple and pair

The output from create_entry() can be processed by any function that operates on tuple, and conversely any function taking the output could also operate on other tuples. But sometimes we want to express *intent*

```
using key_value = std::tuple<int, int>;

auto create_entry() {
  int key{};
  int value{};
  // ...
  // stuff happens
  // ...
  return key_value{key, value};
}
```

Using tuple and pair

Of course, we may prefer to make the function return type explicit

```
using key_value = std::tuple<int, int>;

key_value create_entry() {
  int key{};
  int value{};
  // ...
  // stuff happens
  // ...
  return ~~key_value~~{key, value}; // Implicit creation of tuple
}
```

Using tuple and pair

It is, however, less pleasant in the consuming code

```
int main() {
  const key_value result = create_entry();
  std::cout << std::get<0>(result)
      << "=" << std::get<1>(result) << '\n';
}
```

What are the semantics of “0” and “1” or, equivalently, of `first` and `second` in `std::pair`?

- It gets worse when tuples are *nested*. I've used production code with three levels of nesting.
- Can we retain the flexibility of the underlying data type while adding the ability to express *intent*?

Using tuple and pair

- We could introduce helper variables to express intent:

```
int main() {
  const key_value result = create_entry();
  const auto& key = std::get<0>(result);
  const auto& value = std::get<1>(result);
  std::cout << key << "=" << value << '\n';
}
```

This is nice and expressive; and it also optimises well as the compiler can easily 'see through' the references.

- The same idea works with std::pair:

```
auto pr = map.insert({k,v});
const auto& iter = pr.first;
const auto& inserted = pr.second;
```

Using tuple and pair

- We could introduce a simple helper **class** to express intent:

```
template <typename T, typename U>
struct Entry
{
  Entry(const std::tuple<T,U> &t) :
    key(std::get<0>(t)), value(std::get<1>(t)) {}
  T key;
  U value;
};

int main() {
  const Entry e = create_entry();
  std::cout << e.key << "=" << e.value << '\n';
}
```

However, this risks adding in additional construction; avoiding this adds additional complexity to the constructors of the class

Using tuple and pair

- Another idiom was to introduce helper variables and use `std::tie`:

  ```
  int main() {
    int key;
    int value;
    std::tie(key, value) = create_entry();
    std::cout << key << "=" << value << '\n';
  }
  ```

  This is equally expressive; and it too optimises well as the compiler can effectively elide the tie.

- One *advantage* is the lack of aliasing – there is no 'spare' named variable
- One *disadvantage* is that the variables of object type may be default constructed and then overwritten

Using tuple and pair

- Since C++17 we have been able to use structured bindings

```
int main() {
  const auto [key, value] = create_entry();
  std::cout << key << "=" << value << '\n';
}
```

This is *approximately* equivalent to the first code I showed:

```
  const std::tuple<int, int> __v = create_entry();
  const auto& key = std::get<0>(__v);
  const auto& value = std::get<1>(__v);
```

Using tuple and pair

- It would be nice to be able to use structured bindings with a variadic tuple (for example inside a template)

  ```
  template <typename... T>
  void foo() {
    const auto [...items] = create_entry<T...>();
    process(items...);
  }
  ```

  This is proposed in P1061 “Structured Bindings can introduce a Pack”, which should be part of C++26

Using tuple and pair

- The `std::tie` solution from C++11 is mostly replaced by structured bindings which I believe are generally superior.
- There are still some places where it could be useful
    - The type of the target variable is not that of the returned data
    - You want to update existing variables with the returned data

```
void foo() {
  auto [key, value] = create_entry();
  // ...
  if (needs_refresh) {
    std::tie(key, value) = create_entry();
  //...
  }
}
```

Using tuple and pair - summary

- There seems to be little benefit in using `pair` in C++23 code
- Using `tuple` is getting easier with structured bindings, CTAD, and some of the other changes coming down the C++ pipeline

The rule(s) of 'N'

- When designing a class, the decisions about providing constructors, destructors, and asignment operators are related.
- From the early days of C++ we have had the “rule of three”, coined by Marshal Cline in 1991:
  - “if a class defines any of the following then it should probably explicitly define all three:
    - destructor
    - copy constructor
    - copy assignment operator”

The rule(s) of 'N'

- A standard example of the rule of three, for an owned buffer:

```
class packet {
  std::size_t len_;
  char *buffer_; // owned by the instance of the class
  // ...
public:
  // various constructors...

  ~packet();
  packet(const packet &rhs);
  packet& operator=(const packet &rhs);

  // other methods ...
};
```

The rule(s) of 'N'

- And a possible implementation of the special member functions:

```
packet::~packet() {
  delete buffer_;
}

packet::packet(const packet &rhs)
  : len_(rhs.len_), buffer_(new char [len_])
{
  memcpy(buffer_, rhs.buffer_, len_);
}

packet::packet& operator=(const packet &rhs) {
  packet{rhs}.swap(*this);
  return *this;
}
```

The rule(s) of 'N'

- When designing a class, the decisions about providing constructors, destructors, and asignment operators are related.
- Of course, in C++11 we added move semantics to the language. So there's now a move constructor and a move assignment operator to consider as well.
- So … we're going need a bigger boat

  N += 2
- How to *implement* these two additional special member function depends on what we choose for the semantics of the **moved-from** state

The rule(s) of 'N'

- The rule of five: if a type ever needs one of the following, then it must have all five:
  - destructor
  - copy constructor
  - move constructor
  - copy assignment operator
  - move assignment operator

The rule(s) of 'N'

- Example of the rule of *five* highlighting the additions

```
class packet {
  char *buffer; // owned by the instance of the class
  // ...
public:
  ~packet();
  packet(const packet &rhs);
  packet(packet &&rhs);
  packet& operator=(const packet &rhs);
  packet& operator=(packet &&rhs);
  // ...
};
```

The rule(s) of 'N'

- Example of the rule of ~~*five*~~ three

```
class packet {
  char *buffer; // owned by the instance of the class
  // ...
public:
  ~packet();
  packet(const packet &rhs);
  //packet(packet &&rhs);
  packet& operator=(const packet &rhs);
  //packet& operator=(packet &&rhs);
  // ...
};
```

- If we don't declare the two extra members then they're **not** declared for us. Attempting to **move** objects of the class for construction or assignment simply calls the **copy** operation

The rule(s) of 'N'

- One possible implementation of the two member functions added by the rule of five:

```
packet::packet(packet &&rhs)
: len_(rhs.len_), buffer_(rhs.buffer_) { rhs.buffer_ = 0; )
```

```
packet& operator=(packet &&rhs) {
  this->swap(rhs);
  return *this;
}
```

- The moved from state should be: “valid but unspecified”
- There is a time-bomb here if we ever end up copy constructing from an object moved from by the move constructor above

The rule(s) of 'N'

- The single responsibility principle suggests we delegate the buffer management to a separate object.

```
class packet {
  std::vector<char> buffer_; // (or whatever)
  // ...
public:
  // various constructors...

  // various methods...
};
```

- Now the sub-object manages the ownership and, in this case, also handles the rule of five **for us**

The rule(s) of 'N'

- If we need, for example, a destructor then we can =default the methods we want.

```
class packet {
  std::vector<char> buffer_; // (or whatever)
  // ...
public:
  // various constructors...

  virtual ~packet() = 0;
  packet(const packet &rhs) = default;
  packet(packet &&rhs) = default;
  packet& operator=(const packet &rhs) = default;
  packet& operator=(packet &&rhs) = default;

};
```

The rule(s) of 'N' - summary

- The best case is when N == 0
- We can, and should, make use of helper objects to avoid the “main” class having responsibility for too many things
- When moving older code to modern C++ it is an easy trap to simply add the “missing” move operations when you might do better making the existing copy operations implicit

Initialization

- There are many ways in C++ to initialise even a simple integer variable:

```
int main() {
  int i = 0;
  int j(0);
  int k{0};

  auto l = 0;
  auto m(0);
  auto n{0};
}
```

- What are the differences between these ways?

Initialization

- There are similar options for initializing variables of a class type:

```
int main() {
  std::string i = "test";
  std::string j("test");
  std::string k{"test"};

  auto l = std::string{"test"};
  auto m(std::string("test"));
  auto n{std::string{"test"}};
}
```

- What are the differences between these ways?

Initialization

- What happens when you want a *default* value?

```
int main() {
  int i;
  int j();
  int k{};

  auto l;
  auto m();
  auto n{};
}
```

Initialization

- What happens when you want a *default* value?

```
int main() {
  int i;      // un-initialised
  int j();    // “most vexing parse”
  int k{};    // fine

  auto l;     // error
  auto m();   // “most vexing parse”
  auto n{};   // error
}
```

- **`j` is a declaration of a `function returning int`**
- **`m` is a declaration of a `function with a deduced return type`**

Initialization

- What happens when you want a *default* value of class type?

```
int main() {
  std::string i;
  std::string j();
  std::string k{};

  auto l = std::string{};
  auto m(std::string());
  auto n{std::string{}};
}
```

Initialization

- What happens when you want a *default* value of class type?

```
int main() {
  std::string i;              // default value
  std::string j();            // “most vexing parse”
  std::string k{};            // default value

  auto l = std::string{};     // default value
  auto m(std::string());      // “most vexing parse”
  auto n{std::string{}};      // default value
}
```

- **`j` is a declaration of a function returning `std::string`**
- **`m` is a declaration of a function with a deduced return type taking an argument of a pointer to a function returning `std:string`**

Initialization

- What happens when you provide the *wrong* datatype?

```
int main() {
  int i = 0.0;
  int j(0.0);
  int k{0.0};

  auto l = 0.0;
  auto m(0.0);
  auto n{0.0};
}
```

Initialization

- What happens when you provide the *wrong* datatype?

```
int main() {
  int i = 0.0;     // truncates
  int j(0.0);      // truncates
  int k{0.0};      // error

  auto l = 0.0;    // l is now a double
  auto m(0.0);     // m ''
  auto n{0.0};     // n ''
}
```

Initialization

- What happens when you provide the wrong data type?

```
int main() {
  std::string i = 'Z';
  std::string j('Z');
  std::string k{'Z'};

  auto l = std:string{'Z'};
  auto m(std::string('Z'));
  auto n{std::string{'Z'}};
}
```

Initialization

- What happens when you provide the wrong data type?

```
int main() {
  std::string i = 'Z';           // error
  std::string j('Z');            // error
  std::string k{'Z'};            // "Z"s

  auto l = std:string{'Z'};      // error
  auto m(std::string('Z'));      // error
  auto n{std::string{'Z'}};      // "Z"s
}
```

Initialization

- Sadly initialization in C++ is complicated and we've not had a great track record at making it simpler …
- Be aware of the pitfalls
- Be careful about “drive-by” changes in the name of consistency

Conclusion

- The phrase “There's more than one way to do it” (aka TMTOWTDI) is true in many mature systems
- It's good to keep up to date with the new ways to do what we already know how to do
- However, there are various tradeoffs to make:
    - Readability (for the developers and maintainers of the code)
    - Availability of features in all your target environments
    - “Invisible” performance costs or benefits
        - Differences in runtime overhead
        - Effects on optimization of the resulting code