Easing into C++ Templates - 2

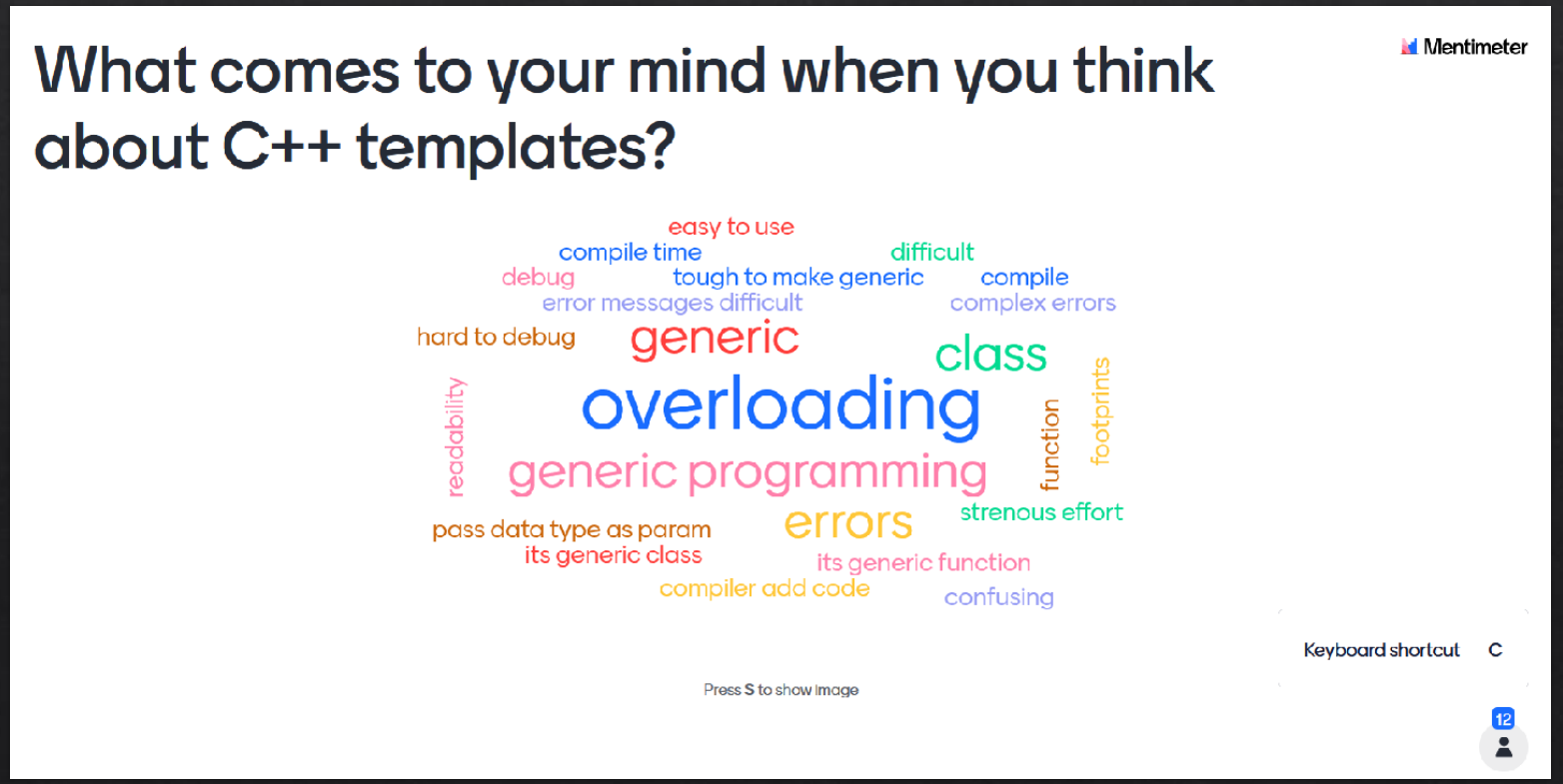
Ankur M. Satle

https://ankursatle.wordpress.com

https://www.linkedin.com/in/ankursatle

In Part 1, you said about templates…

- Generic function, programming
- Compile-time, compiler adds code, type as param
- Overloading
- Footprint
- Generic class
- Readability
- Tough to make generic
- Strenuous effort
- Hard to debug
- Complex/difficult errors



In Part 1, we covered

- Functions :: reuse → Function templates :: Super-reuse (generic)
- Compiler instantiates functions with Parameterized types
  - Real-life example **`process`** with Event containers
- How instantiation works
  - Instantiation in translation unit → binary (keeping just one equivalent copy)
- Explicit type specification & automatic type deduction
  - Compiler deduces correct types
  - Specify types only when needed – avoid
- Generic functions with **`auto`** parameters
- Function templates with multiple parameterized types
- Wanted to finish with one aspect explicitly
- And wanted to remind that…

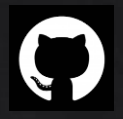
“If you know any programming,
you already know Generic Programming
with `templates`”

Says who?

Ankur M. Satle ☺

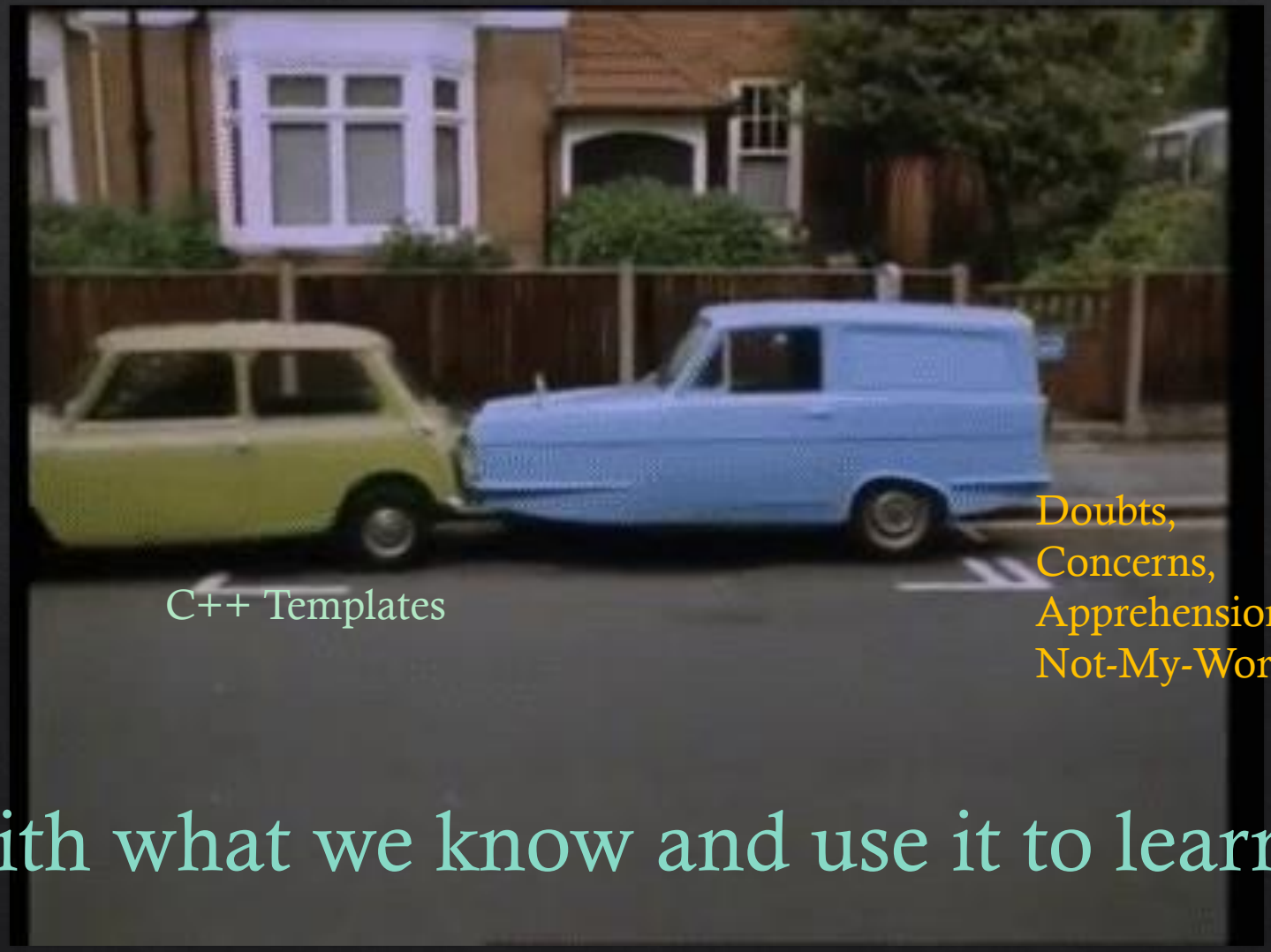
$ who am i

- Modern C++ enthusiast & evangelist
- Extensive Telecom product experience
- Aim for correct by construction implementations
- Architect at EXFO
- Lightning Talk at cppcon 2020 on "Why C++ for Large Scale Systems" – had only 5 min 😐
- Actively working on Cloud Native, CI/CD, security
- https://ankursatle.wordpress.com



- My friends claim that I cannot write code without `template<typename T>` ☺
- I appreciate the concerns of C and C++98 style programmers around templates
- I fairly appreciate what how templates can be helpful

Easing into C++ Templates then…



We start with what we know and use it to learn templates

Templates save us from donkey work

from implementing all possible type combinations

Multiple data-types – combinations explode!

- Consider the following function template

```
template<typename InputIt,
typename OutputIt>

OutputIt copy(InputIt in,
InputIt end, OutputIt out) {
   while (in != end) {
      *out++ = *in++;
   }
   return out;
}
```

- Imagine implementing copy for possibile `InputIt` and `OutputIt` combination!
- For 3 data-types, the no of ways copy could be needed is ${}^3P_2 = 6$
- For 4 data-types, the no. of ways copy may be needed is ${}^4P_2 = 12$
- Writing all these by hand → error-prone, tedious, wastage of human effort
- Nobody will write these many, I hear… the same code will be copy-pasted and reimplemented as and when needed. This is losing speed of delivery!

Multiple data-types – combinations explode!

- Consider the following function template

```
template<typename InputIt,
typename OutputIt>

OutputIt copy(InputIt in,
InputIt end, OutputIt out) {
   while (in != end) {
      *out++ = *in++;
   }
   return out;
}
```

- Code generated by compiler → correct by default → Human effort saved!
- Generated binary has only instances that are used
- `inline` eliminates function calls
- Win-win → small footprint

“

Life is better with Generic Programming
and `templates`

”

Ankur M. Satle ☺

If you understand polymorphism
you can work with templates

How C deals with polymorphism

- Let's look at **<stdlib.h>** quick sort function:

```
errno_t qsort_s( void *ptr, rsize_t count, rsize_t size,
int (*comp)(const void *, const void *, void *), void *context );
```

- Here, you could be sorting any user-defined data-type
- **ptr** is a **void*** → all knowledge of actual type is lost
- Programmer must provide the size of item (**size**) to the compiler!
- **qsort_s** deals with items in terms of memory bytes rather than intelligently
- Comparison is abstracted via **comp** function
- Context is an argument passed on each invocation of **comp**

C++ Inheritance-based Polymorphic Code

```
class Message {
   virtual string as_string() = 0;
};
class Request : public Message {
   string as_string() override {…}
};
class Response : public Message {
   string as_string() override {…}
};
ostream& operator<<(ostream& out,
const Message& msg) {
   out << msg.as_string();
   return out;
}
```

- This enables code like this:

```
Message* msg = new Request();
json_ostream << *msg;
```

- Dynamic Polymorphism = virtual dispatch
- Indirect function call, instruction cache miss = overheads

C++ Polymorphism using templates

```
class Request ~~: public Message~~ {
    string as_string() ~~override~~ {…}
};
class Response ~~: public Message~~ {
    string as_string() ~~override~~ {…}
};
template<typename T>
ostream& operator<<(ostream& out,
const T& msg) {
    out << msg.as_string();
    return out;
}
```

- User code can deal with concrete types:

```
auto req = new Request();
json_ostream << *req;
```

- Works with unrelated data-types
- No virtual dispatch overhead
- Functions can be `inlined` eliminating the function call altogether

C++ Polymorphism using templates

```
ostream& operator<<(ostream& out,
const Base& obj) {
   out << to_string(obj);
   return out;
};
```

vs.

```
template<typename T>
ostream& operator<<(ostream& out,
const T& t) {
   out << to_string(t);
   return out;
}
```

- Usage:

```
auto obj = new Derived();
std::clog << *obj;
auto player = new Monster();
std::clog << player;    //Oops
```

- References to `Base` results in splicing
- Templates use the correct type
- Implementing for `Base` did not help similar usage of `player`

C++ Polymorphism using templates

```
ostream& operator<<(ostream& out,
const Base& obj) {
    out << to_string(obj);
    return out;
};
```

vs.

```
template<typename T>
ostream& operator<<(ostream& out,
const T& t) {
    out << to_string(t);
    return out;
}
```

- Usage:

```
int num = 42;
std::clog << num;
```

- Implementing for `Base` did not help basic datatypes like `int`
- `int` cannot be enforced to be derived from `Base`
- Templates enable generic code for primitive data-types = no problem!

C++ Inheritance-based Polymorphic Code

```
class Message {
   virtual string as_string() = 0;
};
class Request : public Message {
   string as_string() override {…}
};
class Response : public Message {
   string as_string() override {…}
};
ostream& operator<<(ostream& out,
const Message& msg) {
   out << msg.as_string();
   return out;
}
```

- What if we needed this?

```
pair<const char*, double> field =
{"pi", 3.1415};
json_ostream << field;
//error: no matching function
```

- **pair** cannot inherit from Message!
- Inheritance is intrusive
- Types may not be modifiable e.g. when provided by 3rd-party/external library

“If you know any programming,
you already know Generic Programming
with `templates`”

Says who?

Ankur M. Satle ☺

No `override`

`override` doesn't help templates

- Inheritance-based:

```
class Message {
   virtual string as_string() = 0;
};
class Request : public Message {
   string as_string(int) override
{…}
};
ostream& operator<<(ostream& out,
const Message& msg) {
   out << msg.as_string();
   return out;
}
```

- Usage code:

```
Message* msg = new Request();
cout << *msg;
```

- This results in compilation error. Can you spot it?

`override` doesn't help templates

- Inheritance-based:

```
class Message {
   virtual string as_string() = 0;
};
class Request : public Message {
   string as_string(int) override
{…}
};
ostream& operator<<(ostream& out,
const Message& msg) {
   out << msg.as_string();
   return out;
}
```

- Usage code:

```
Message* msg = new Request();
cout << *msg;
```

- Error immediately pointed out for Request::as_string() due to incorrect signature
- Thanks to `override`

`override` doesn't help templates

- Template-based:

```
class Request {
   string as_string(int) {…}
};
template<typename T>
ostream& operator<<(ostream&
out, const T& msg) {
   out << msg.as_string();
   return out;
}
```

- Usage code:

```
auto req = new Request();
cout << *req;
```

- Error pointed out when template function is instantiated by compiler for Request
- Enter compilation errors for templates – which can get difficult to understand!

Class templates

We may need to handle parameterized classes

- Containers are a classic example:

```
class VectorOfInts {
   unique_ptr<int[]> data;
   size_t size;
}

class VectorOfStrings {
   unique_ptr<string[]> data;
   size_t size;
}
```

- Usage:

```
VectorOfInts ivec = {i, j, k};
VectorOfStrings svec =
   {name, middle, surname};
cout << ivec.size() << ivec.back();
cout << svec.size() << svec.back();
```

- Looks like donkey-work!

Parameterized class

- Let's templatise

```
template<typename T>
class vector {
   unique_ptr<T[]> data;
   size_t size;
}
```

- Can we say then:

  `template` = smart work?

- Usage:

```
vector<int> ivec = {i, j, k};
vector<string> svec =
   {name, middle, surname};
cout << ivec.size() << ivec.back();
cout << svec.size() << svec.back();
```

- Now, we can also handle other types:

```
vector<Event> evts = {call, ring,
accept, reject, hang_up};
evts.push_back(dnd);
cout << evts.size() << evts.back();
```

Where do templates help us?

- Algorithms: `sort`, `copy`, `binary_search`, `reduce`
- Utility functions: `swap`, `exchange`
- Function objects: `function`, `invoke`, `plus`, `not_equal_to`, `bind`
- Utility classes: `pair`, `tuple`, `hash`
- Containers: `vector`, `array`, `list`, `set`, `multimap`, `unordered_map`
- Date and Time: `time_point`, `duration`, `zoned_time`

- But wait… only standard library constructs?
- Are templates not usable beyond these!?

Consider

- A chat application (just like WhatsApp)
- Enables sending & receiving messages
- Over some connection
- Using a protocol (SIP or HTTP)

The Sender

- Consider a Sender class

```
class Sender {
   pair<chatid, msgid> send(string remote, string msg);
   msgid send(chatid id, string msg);
   int socket_fd;
};
```

- Usage:

```
Sender sender( /* Server details */ );
auto [cid, mid] = sender.send("+91-99887-11223", "Hi");
```

Sender analysis

- The Sender seems to be doing too much!
- It receives the end-user message string
- Has a network socket descriptor
- Can we say the send function:
  - Encodes the message into a network message
  - Sends it over a network connection
  - Gets back a response (with chatid & msgid)
- This is too much functionality

Message encoding

- How about splitting concerns?

```
class SipEncoder {
   encoded_msg encode(string remote, string msg) {
      return encode_headers(remote) + encode_body(msg);
   }
};
```

- Usage:

```
SipEncoder encoder;
auto network_msg = encoder.encode("+91-99887-11223", "Hi");
```

Network Connection

- How about splitting concerns?

```
class TcpConnection {
   void write(span<const byte> msg) {
      //Socket operations
   }
   int socket_fd;
};
```

- Usage:

```
TcpConnection conn;
encoded_msg network_msg = …;
conn.write(network_msg);
```

A better Sender

```
template<typename Encoder, typename Connection>
class Sender {
   pair<chatid, msgid> send(string remote, string msg);
   msgid send(chatid id, string msg);
   Encoder encoder;
   Connection conn;
};
```

- What would member functions look like?

A better Sender::send implementation

- A better Sender::send

```
template<typename Encoder, typename Connection>
pair<chatid, msgid> Sender<Encoder, Connection>::send(
        string remote, string msg) {
    auto net_msg = encoder.encode(remote, msg);
    conn.write(net_msg);
    //get chatid & msgid for the message
    return {cid, mid};
};
```

- Member functions can be defined out of the class body but must be in the header

Using the better Sender

- Using the better Sender

```
auto sender = Sender<SipEncoder, TcpConnection>(/*Server details*/);
auto [remote, msg] = get_user_input();
auto [cid, mid] = sender.send(remote, msg);
auto another_msg = get_user_input();
auto another_mid = sender.send(cid, another_msg);
```

- The using code is almost the same

The better Sender is ready for future

- Now, app must send secured messages (remember end-to-end encryption in WhatsApp?)

```
auto sender = Sender<SipEncoder, TlsConnection>(/*Server details*/);
auto [remote, msg] = get_user_input();
auto [cid, mid] = sender.send(remote, msg);
auto another_msg = get_user_input();
auto another_mid = sender.send(cid, another_msg);
```

- By just introducing `TlsConnection`, the same code sends encrypted messages!
- A big step towards code stability (less defects) & high reuse (effort saving)
- Templates are a customization point

Help readability

- With template parameters, types → quite a lineful and mouthful ☺

```
using SecuredSender = Sender<SipEncoder, TlsConnection>;
```

- Now, usage code changes from

```
Sender<SipEncoder, TlsConnection> sender(/*Server Details*/);
```

- to

```
SecuredSender sender(/*Server Details*/);    //Looks peaceful!
SharedResource res;  //Instead of Resource<vector<Event>, mutex>
```

- `using` declarations help to not get lost in <angle bracket jungle>

“

Life is better with Generic Programming
and `templates`

”

Ankur M. Satle ☺

Templates help with testability

Dependency injection

Templates help with testability

```
auto s = Sender<MockEncoder, MockConnection>(/*Server details*/);
REQUIRE(s.send("+91-99887-11223", "Hi"), make_pair(1, 100));//Catch2
REQUIRE(s.send(1, "Bye"), 101);
```

- By using **MockEncoder** and **MockConnection**, we can test Sender controlling behaviour from the Mock classes!
- Moreover, **SipEncoder** and **TcpConnection** are also independently testable
- Test coverage goes up very high as it is not a big ball of mud
- Every individual part is independent & simple to understand. Less complexity!
- Reuse++ for every individual concept

“

Life is better with Generic Programming
and `templates`

”

Ankur M. Satle ☺

Member Function Template

`operator==` on a class

- Consider a class for Email-id

```
class EmailId {
   bool operator==(
      const std::string& id);

   string display_name;
   string username;
   string domain;
};
```

- Usage:

```
EmailId id = emp.emailid();
string user_id = get_winner();
if (id == user_id) {
   award(emp);
}
```

- But is matching only with std::string good enough?
- Matching with `const char*` and `std::string_view` are natural expectations

Template to support more types

- Function templates are possible
- And Member Function Templates?

```
class EmailId {
   template<typename StrLike>
   bool operator==(
      const StrLike& id);

   string display_name;
   string username;
   string domain;
};
```

- Usage:

```
EmailId id = emp.emailid();
const char* user_id = get_query_id();
if (id == user_id) {
   display(emp);
}
```

- Member Function Templates help introduce a type within a member function
- Class may or may not be templated itself
- Class template types & Member function template types are both available within the body of the member function

“

Life is better with Generic Programming
and `templates`

”

Ankur M. Satle ☺

In conclusion of Part 2

- Templates are great!
- Templates:
  - Create instantiation for every possible type combination
  - Produce optimized code
  - Enable building utilities, containers, type_traits
  - Help with higher level code in keeping functional separation between concerns
  - Assist with testability
- And just to remind you before we end…

- Templates are, indeed, the answer to so many questions!

There is more to templates…

- Dependent types
- Non-type template parameters
- Function specialization
- Class specialization
- Explicit template instantiation
- Template errors
- Variadic templates
- Working with forwarding references
- Policy-based design

“

Life is better with Generic Programming
and `templates`

”

Ankur M. Satle ☺

Questions?

Thank you!

See you in the next part of this discussion

Ankur M. Satle