Easing into C++ Templates - 1

Ankur M. Satle

https://ankursatle.wordpress.com

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

Easing into C++ Templates with real-world examples

The title in full ☺

C++ Templates
for those who do not need them

Alternate title

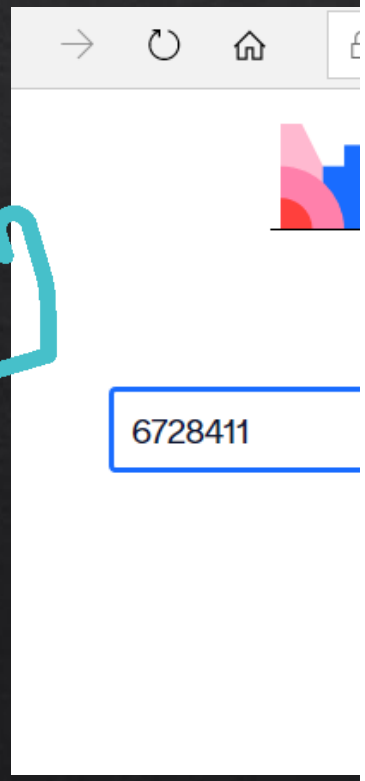
What comes to your mind when you see…

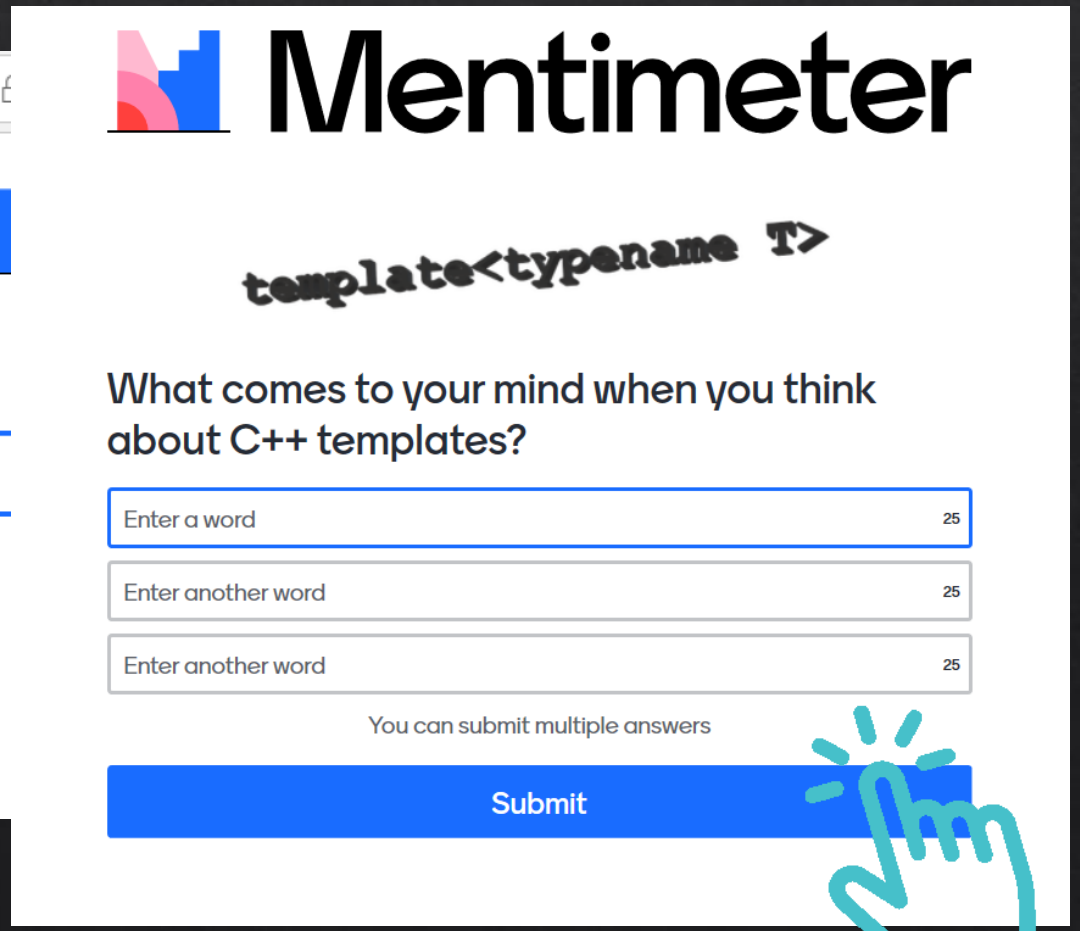
`template<typename T>`

Open www.menti.com

Enter code 67 28 411

Submit your thoughts

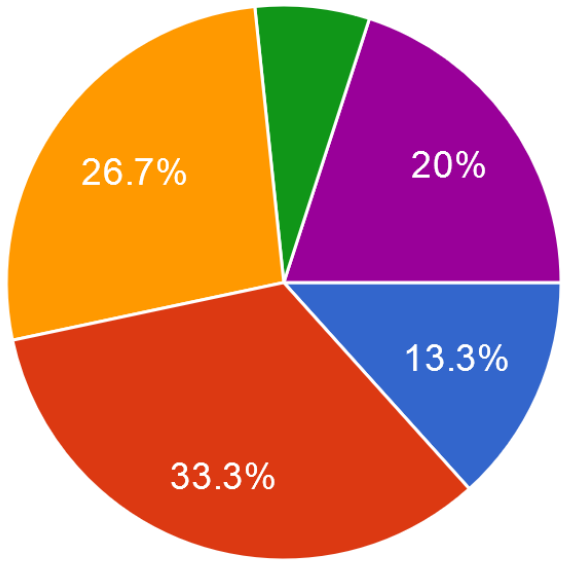




What you told us previously

How comfortable are you with template programming in C++?

15 responses

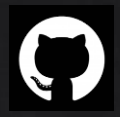


- I stay away from templates (say it if this is the case ;-))
- I have used C++ standard & 3rd-party libraries but not heavily
- I have created a few templates
- I work with heavily templated & generic code
- I can template meta-program

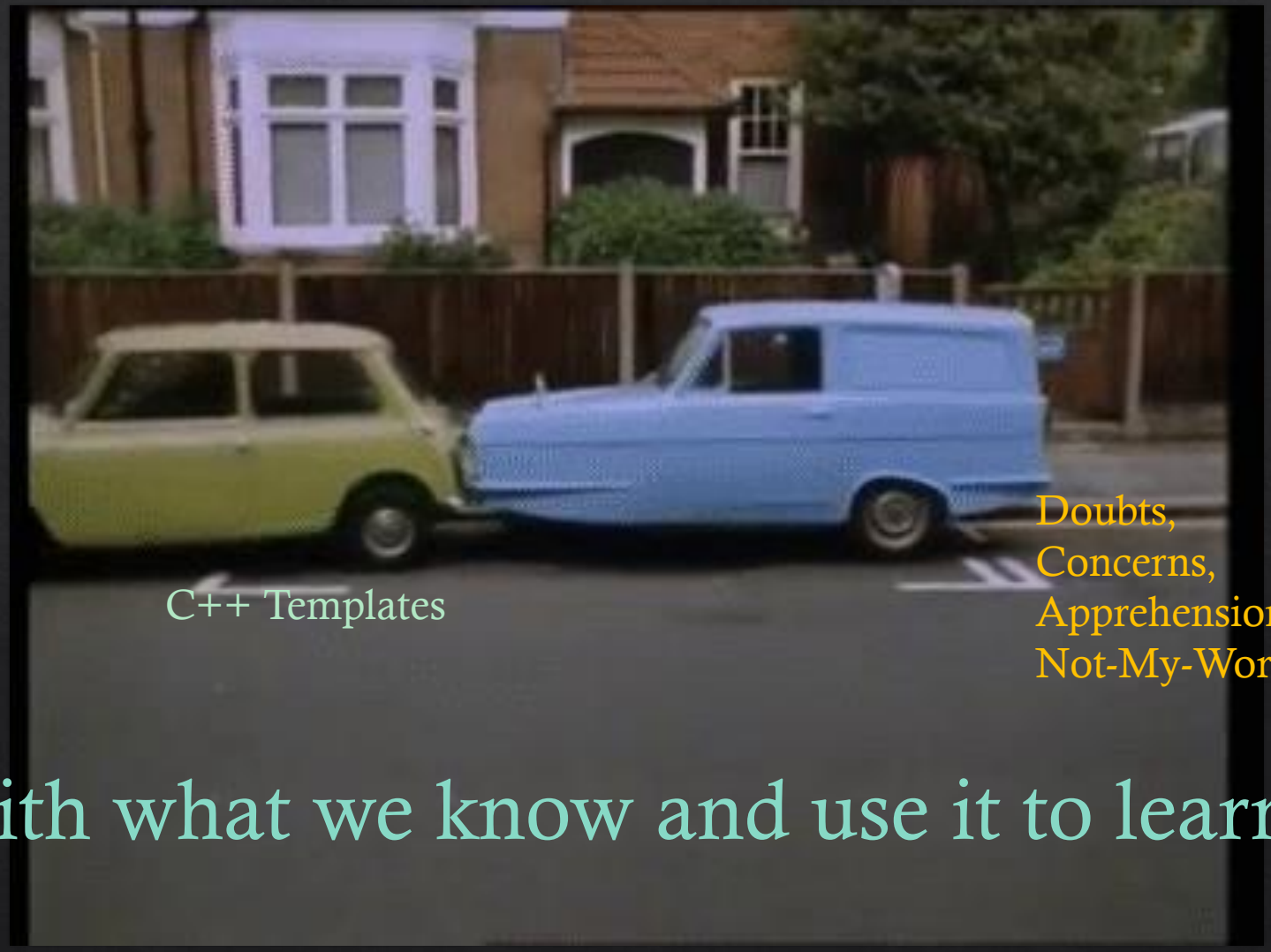
$ who am i

- 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, Real-time packet streaming
- 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 are good at and help us



Easing into C++ Templates then…



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

Spoiler Alert…

But first…

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

Says who?

Ankur M. Satle ☺

Generic Programming

Abstractions

- Values → variables

```
X = X + 1
```

- Code → small units → functions
- Pass any value and reap the benefit

```
Fx = FAERO(2.5, 12.5, 100.0, 23.3)
```

```
Fy = FAERO(2.5, 69.1, 88.0, 125.0)
```

- Data-types → templates

```
template<typename T>
```

```
T sum(const T& x, const T& y);
```

```
C      Aerodynamics (1975)
C      P. 50
C      Aerodynamic Force
C      CF   : Force Coefficient (Lift, Drag, Normal Force, etc.)
C      RHO  : Atmospheric Density
C       V   : Air Velocity
C       S   : Surface Area
       REAL FUNCTION FAERO (CF, RHO, V, S) RESULT (F)
       REAL CF, RHO, V, S, F
       REAL*8 Q
       Q = RHO*(V**2)*S / 2.0          !dynamic pressure
       F = CF*Q
       RETURN
       END FUNCTION FAERO
```

How does C deal with different data-types?

- A simple function abs() that deals with ints may be implemented like this:

```
int abs(int n) {
    return n < 0? -n: n;
}
```

- Clearly not generic!
- Code changes with data-types
- Functions must be explicitly named!

- `<stdlib.h>`, `<math.h>` & `<complex.h>` have:

```
int abs(int n);
long labs(long n);
long long llabs(long long n);
intmax_t imaxabs(intmax_t n);
float fabsf(float n);
double fabs(double n);
long double fabsl(long double n);
float cabsf(float complex z);
double cabs(double complex z);
long double cabsl(long double complex z);
```

C++ - Function Overloading - `swap`

- Consider two functions:

```
void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

void swap(string& a, string& b) {
    string temp = a;
    a = b;
    b = temp;
}
```

- Usage:

```
int i, j;
string name, surname;
swap(i, j);
swap(name, surname);
swap(name, j);  //error: no matching function
```

- Overloading enables generic code

Function Overloading – work with containers

```
void process(std::vector<Event>& events) {
    for (auto& ev : events) {
        process(ev);
    }
}

void process(std::list<Event>& events) {
    for (auto& ev : events) {
        process(ev);
    }
}
```

- Usage:

```
vector<Event> evs =
read_from_network();
process(evs);


list<Event> ev_list =
read_from_file();
process(ev_list);


array<Event, 4> init_evs =
{load, init, disp, alert};
process(init_evs);
```

Function Templates

```
void process(std::vector<Event>& events) {
    for (auto& ev : events) {
        process(ev);
    }
}

void process(std::list<Event>& events) {
    for (auto& ev : events) {
        process(ev);
    }
}
```

- Usage:

```
vector<Event> evs =
read_from_network();
process(evs);


list<Event> ev_list =
read_from_file();
process(ev_list);


array<Event, 4> init_evs =
{load, init, disp, alert};
process(init_evs);
```

Function Templates – typename T

```
template<typename T>
void process(T& events) {
    for (auto& ev : events) {
        process(ev);
    }
}
```

- T is a Container

- Usage:

```
vector<Event> evs =
read_from_network();
process(evs);


list<Event> ev_list =
read_from_file();
process(ev_list);


array<Event, 4> init_evs =
{load, init, disp, alert};
process(init_evs);
```

Function Templates – `typename T`

```
template<typename T>
void process(T& events) {
    for (auto& ev : events) {
        process(ev);
    }
}
```

- Generic code can deal with data-types never seen/imagined before
- Makes implementations more future-ready

- Usage:

```
vector<Event> evs =
read_from_network();
process(evs);


list<Event> ev_list =
read_from_file();
process(ev_list);


array<Event, 4> init_evs =
{load, init, disp, alert};
process(init_evs);
```

Function Templates – Better named

```
template<typename Container>
void process(Container& events) {
    for (auto& ev : events) {
        process(ev);
    }
}
```

- Template arguments (parameters) need not be named T
- Meaningful names aid better understanding

- Usage:

```
vector<Event> evs =
read_from_network();
process(evs);


list<Event> ev_list =
read_from_file();
process(ev_list);


array<Event, 4> init_evs =
{load, init, disp, alert};
process(init_evs);
```

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

Says who?

Ankur M. Satle ☺

Function template instantiation

Function Templates – Instantiation

- Given:

```
template<typename Container>
void process(Container& events) {
    for (auto& ev : events) {
        process(ev);
    }
}
```

- Compiler instantiates a template function:

```
void process(std::vector<Event>& events) {
    for (auto& ev : events) {
        process(ev);
    }
}
```

- When compiler sees this code:

```
vector<Event> evs =
read_from_network();
process(evs);
```

- Instantiation when compiler encounters usage with a new set of data-types
- New template function (instance) is stamped out
- If already instantiated, it is used

Function Templates – Instantiation

- Given:

```
template<typename Container>
void process(Container& events) {
    for (auto& ev : events) {
        process(ev);
    }
}
```

- Compiler instantiates a template function:

```
void process(std::list<Event>& events) {
    for (auto& ev : events) {
        process(ev);
    }
}
```

- When compiler sees this code:

```
list<Event> ev_list =
read_from_file();
process(ev_list);
process(ev_list2);
process(ev_list3);
```

- Every translation unit → compile- time optimization → link-time optimization → ODR → one copy for every type-combination

Function Templates – Instantiation

- Given:

```
template<typename Container>
void process(Container& events) {
    for (auto& ev : events) {
        process(ev);
    }
}
```

- Compiler instantiates a template function:

```
void process(std::array<Event>& events) {
    for (auto& ev : events) {
        process(ev);
    }
}
```

- When compiler sees this code:

```
array<Event, 4> init_evs =
{load, init, disp, alert};
process(init_evs);
```

- Works like a charm for new data-types too!
- Templates get you ready for types not handled before

Specify types or let compiler deduce?

Explicitly specify types or deduce the types

- Consider the following function template

```
template<typename T>
T sum(T a, T b) {
   return a + b;
}
```

- Guess the data-types:

```
auto total1 = sum(2, 3);
auto total2 = sum(9.8, 3.14);
auto lowercase = sum('A', 32);
```

- The deduced type should agree for every type
- Type conversion is not allowed with templated parameters

Explicitly specify types or deduce the types

- Consider the following function template

```
template<typename T>
T sum(T a, T b) {
   return a + b;
}
```

- Guess the data-types:

```
auto total1 = sum<int>(2, 3);
auto total2 = sum<double>(9.8, 3.14);
auto lowercase = sum<char>('A', 32);
```

Special cases

- Consider the following function template

```
template<typename T>
T sum(T a, T b) {
   return a + b;
}
```

- Choices:

```
auto lowercase = sum ('A',
static_cast<char>(32));
```

Or

```
auto lowercase = sum<char>('A', 32);
```

- Let automatic deduction take the calls! ☺
- Specify the data-type to make it explicit that you are taking manual control!

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

Says who?

Ankur M. Satle ☺

Use `auto` for parameter types?

Why not use C++20 `auto` parameters?

- How about changing the following

```
template<typename T>
T sum(T a, T b) {
   return a + b;
}
```

Why not use C++20 `auto` parameters?

- to this?

```
auto sum(auto a, auto b) {
    return a + b;
}
```

- These are called as generic functions as per the standard

Why not use C++20 `auto` parameters?

- For this generic function:

```
auto sum(auto a, auto b) {
   return a + b;
}
```

- Can specify explicit types for auto parameters
- I suggest specifying all types
- I didn't know types for auto parameters could be specified till I tried this out

- Guess the data-types now:

```
auto total1 = sum(2, 3);
auto total2 = sum(9.8, 3.14);
auto lowercase = sum('A', 32);
auto lc1 = sum<char>('A', 32);
//This calls: int sum<char, int>
auto lc2 = sum<char, char>('A', 32);
//Both types are forced to be char,
also returns char
```

Why not use C++20 `auto` parameters?

- How about writing the following as this?

```
auto sum(auto a, auto b) {
   return a + b;
}
```

- Absolutely anything goes!
- Effectively, there is no constraint at all – this is the `void*` of C++
- This gets you in the realm of the infamous template errors!

- Guess the data-types now:

```
auto cpp20 = sum("c++"s, 20);
auto oops = sum("c++", 20);
Employee emp1;
Department dept1;
auto huh = sum(emp1, dept1);
```

```
error: no match for 'operator+'
(operand types are 'Employee' and
'Department')
```

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

Says who?

Ankur M. Satle ☺

Multiple template parameters

What works for one, can work for others too!

Multiple data-types

- 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;
}
```

- This enables us to write:

```
vector<uint8_t> bytes = generate();
auto msg = std::make_unique<
uint8_t[]>(bytes.size());
copy(begin(bytes), end(bytes),
msg.get());
send(std::move(msg));
```

- `InputIt = vector<uint8_t>::iterator`
- `OutputIt = uint8_t*`

Multiple data-types

- 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;
}
```

- Now, we can write this too:

```
const int size = 1024;
uint8_t* bytes = generate(size);
auto msg = vector<uint8_t>(size);
copy(bytes, bytes + size,
back_inserter(msg));
send(msg);
```

- `InputIt = uint8_t*`
- `OutputIt = vector<uint8_t>::iterator`

Multiple data-types

- 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;
}
```

- Oops, sh… happens:

```
const int size = 1024;
uint8_t* bytes = generate(size);
auto msg = vector<uint8_t>(size);
copy(bytes, size, begin(msg));
error: no matching function
send(msg);
```

- InputIt = uint8_t* //As per param 1
- InputIt = int //As per param 2
- OutputIt = vector<uint8_t>::iterator

“

Life is better with Generic Programming
and `templates`

”

Says who?

Ankur M. Satle ☺

Templates save us from having to implement 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 ${}^4P_2 = 12$
- You get the point, writing all these by hand is 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!

“

Life is better with Generic Programming
and `templates`

”

Says who?

Ankur M. Satle ☺

In conclusion of Part 1

- Templates are a great!
- Templates:
  - Are Type-safe
  - Help create generic reusable constructs
  - Instantiate correct code for different types!
  - Reduce manual effort
  - Produce optimized code
  - auto types help create generic functions
- And one last time for today…

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

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

Says who?

Ankur M. Satle ☺

Questions?

Thank you!

See you in the next part of this discussion

Ankur M. Satle