C++ templates Turing-complete?

Asked 16 years, 2 months ago Modified 3 years ago Viewed 51k times

143

I'm told that the template system in C++ is Turing-complete at compile time. This is mentioned in this post and also on wikipedia.

Can you provide a nontrivial example of a computation that exploits this property?

Is this fact useful in practice?

c++ templates template-meta-programming turing-complete

Share

Improve this question

Follow

edited May 23, 2017 at 11:47

Community Bot

1 • 1

asked Oct 9, 2008 at 20:53

Federico A. Ramponi

47k • 30 • 111 • 133

1 "C++ Templates Are Turing Complete" gives an implementation of a Turing machine in templates ... which is non-trivial and proves the point in a very direct way. Of course, it also isn't very useful! – Rob Walker Oct 9, 2008 at 20:57

2 You can check this article from Dr. Dobbs on a FFT implementation with templates which I think not that trivial. The main point is to allow the compiler to perform a better optimization than for non template implementations as the FFT algorithm uses a lot of constants (sin tables for instance) part I part II – florentazcarate Oct 10, 2008 at 12:53

12 Answers

Sorted by: Highest score (default)

235

I've done a turing machine in C++11. Features that C++11 adds are not significant for the turing machine indeed. It just provides for arbitrary length rule lists using variadic templates, instead of using perverse macro metaprogramming :). The names for the conditions are used to output a diagram on stdout. i've removed that code to keep the sample short.

```
#include <iostream>

template<bool C, typename A, typename B>
struct Conditional {
    typedef A type;
};

template<typename A, typename B>
struct Conditional<false, A, B> {
    typedef B type;
};

template<typename...>
struct ParameterPack;

template<bool C, typename = void>
struct EnableIf { };

template<typename Type>
```

```
struct EnableIf<true, Type> {
    typedef Type type;
};

template<typename T>
struct Identity {
    typedef T type;
};

// define a type list
template<typename...>
struct TypeList;

template<typename T, typename... TT>
struct TypeList<T, TT...>  {
    typedef T type;
    typedef TypeList<TT...> tail;
};

template<>
struct TypeList<> {

};

template<typename List>
struct GetSize;

template<typename... Items>
struct GetSize<TypeList<Items...>> {
    enum { value = sizeof...(Items) };
};

template<typename... T>
struct ConcatList;

template<typename... First, typename... Second, typena
struct ConcatList<TypeList<First...>, TypeList<Second.
    typedef typename ConcatList<TypeList<First..., Sec
                                Tail...>::type type;
};

template<typename T>
struct ConcatList<T> {
    typedef T type;
```

```
};

template<typename NewItem, typename List>
struct AppendItem;

template<typename NewItem, typename...Items>
struct AppendItem<NewItem, TypeList<Items...>> {
    typedef TypeList<Items..., NewItem> type;
};

template<typename NewItem, typename List>
struct PrependItem;

template<typename NewItem, typename...Items>
struct PrependItem<NewItem, TypeList<Items...>> {
    typedef TypeList<NewItem, Items...> type;
};

template<typename List, int N, typename = void>
struct GetItem {
    static_assert(N > 0, "index cannot be negative");
    static_assert(GetSize<List>::value > 0, "index too
    typedef typename GetItem<typename List::tail, N-1>
};

template<typename List>
struct GetItem<List, 0> {
    static_assert(GetSize<List>::value > 0, "index too
    typedef typename List::type type;
};

template<typename List, template<typename, typename...
typename... Keys>
struct FindItem {
    static_assert(GetSize<List>::value > 0, "Could not
    typedef typename List::type current_type;
    typedef typename Conditional<Matcher<current_type,
                                 Identity<current_type
                                 FindItem<typename Lis
Keys...>>
        ::type::type type;
};

template<typename List, int I, typename NewItem>
```

```
struct ReplaceItem {
    static_assert(I > 0, "index cannot be negative");
    static_assert(GetSize<List>::value > 0, "index too
    typedef typename PrependItem<typename List::type,
                                 typename ReplaceItem<type
                                                      NewI
        ::type type;
};

template<typename NewItem, typename Type, typename...
struct ReplaceItem<TypeList<Type, T...>, 0, NewItem> {
    typedef TypeList<NewItem, T...> type;
};

enum Direction {
    Left = -1,
    Right = 1
};

template<typename OldState, typename Input, typename N
         typename Output, Direction Move>
struct Rule {
    typedef OldState old_state;
    typedef Input input;
    typedef NewState new_state;
    typedef Output output;
    static Direction const direction = Move;
};

template<typename A, typename B>
struct IsSame {
    enum { value = false };
};

template<typename A>
struct IsSame<A, A> {
    enum { value = true };
};

template<typename Input, typename State, int Position>
struct Configuration {
    typedef Input input;
    typedef State state;
    enum { position = Position };
```

```
};

template<int A, int B>
struct Max {
    enum { value = A > B ? A : B };
};

template<int n>
struct State {
    enum { value = n };
    static char const * name;
};

template<int n>
char const* State<n>::name = "unnamed";

struct QAccept {
    enum { value = -1 };
    static char const* name;
};

struct QReject {
    enum { value = -2 };
    static char const* name;
};

#define DEF_STATE(ID, NAME) \
    typedef State<ID> NAME ; \
    NAME :: name = #NAME ;

template<int n>
struct Input {
    enum { value = n };
    static char const * name;

    template<int... I>
    struct Generate {
        typedef TypeList<Input<I>...> type;
    };
};

template<int n>
char const* Input<n>::name = "unnamed";
```

```
typedef Input<-1> InputBlank;

#define DEF_INPUT(ID, NAME) \
    typedef Input<ID> NAME ; \
    NAME :: name = #NAME ;

template<typename Config, typename Transitions, typena
struct Controller {
    typedef Config config;
    enum { position = config::position };

    typedef typename Conditional<
        static_cast<int>(GetSize<typename config::inpu
            <= static_cast<int>(position),
        AppendItem<InputBlank, typename config::input>
        Identity<typename config::input>>::type::type
    typedef typename config::state state;

    typedef typename GetItem<input, position>::type ce

    template<typename Item, typename State, typename C
    struct Matcher {
        typedef typename Item::old_state checking_stat
        typedef typename Item::input checking_input;
        enum { value = IsSame<State, checking_state>::
                       IsSame<Cell,  checking_input>::
        };
    };
    typedef typename FindItem<Transitions, Matcher, st

    typedef typename ReplaceItem<input, position, type
new_input;
    typedef typename rule::new_state new_state;
    typedef Configuration<new_input,
                          new_state,
                          Max<position + rule::directi
new_config;

    typedef Controller<new_config, Transitions> next_s
    typedef typename next_step::end_config end_config;
    typedef typename next_step::end_input end_input;
    typedef typename next_step::end_state end_state;
    enum { end_position = next_step::position };
};
```

```
template<typename Input, typename State, int Position,
struct Controller<Configuration<Input, State, Position
                  typename EnableIf<IsSame<State, QAcc
                                    IsSame<State, QRej
    typedef Configuration<Input, State, Position> conf
    enum { position = config::position };
    typedef typename Conditional<
        static_cast<int>(GetSize<typename config::inpu
            <= static_cast<int>(position),
        AppendItem<InputBlank, typename config::input>
        Identity<typename config::input>>::type::type
    typedef typename config::state state;

    typedef config end_config;
    typedef input end_input;
    typedef state end_state;
    enum { end_position = position };
};

template<typename Input, typename Transitions, typenam
struct TuringMachine {
    typedef Input input;
    typedef Transitions transitions;
    typedef StartState start_state;

    typedef Controller<Configuration<Input, StartState
controller;
    typedef typename controller::end_config end_config
    typedef typename controller::end_input end_input;
    typedef typename controller::end_state end_state;
    enum { end_position = controller::end_position };
};

#include <ostream>

template<>
char const* Input<-1>::name = "_";

char const* QAccept::name = "qaccept";
char const* QReject::name = "qreject";

int main() {
    DEF_INPUT(1, x);
```

```
    DEF_INPUT(2, x_mark);
    DEF_INPUT(3, split);

    DEF_STATE(0, start);
    DEF_STATE(1, find_blank);
    DEF_STATE(2, go_back);

    /* syntax:  State, Input, NewState, Output, Move *
    typedef TypeList<
        Rule<start, x, find_blank, x_mark, Right>,
        Rule<find_blank, x, find_blank, x, Right>,
        Rule<find_blank, split, find_blank, split, Rig
        Rule<find_blank, InputBlank, go_back, x, Left>
        Rule<go_back, x, go_back, x, Left>,
        Rule<go_back, split, go_back, split, Left>,
        Rule<go_back, x_mark, start, x, Right>,
        Rule<start, split, QAccept, split, Left>> rule

    /* syntax: initial input, rules, start state */
    typedef TuringMachine<TypeList<x, x, x, x, split>,
    static_assert(IsSame<double_it::end_input,
                         TypeList<x, x, x, x, split, x
                "Hmm... This is borky!");
}
```

Share Improve this answer

edited Jan 26, 2015 at 17:09

Follow

answered Nov 8, 2008 at 22:18

Johannes Schaub - litb

506k ● 131 ● 917 ● 1.2k

194 You have way too much time on your hands. – Mark Kegel Dec 13, 2008 at 19:41

4 It looks like lisp except with a certin word replacing all those parentheses. – Simon Kuang Jun 6, 2014 at 5:44

1 Is the full source publicly available somewhere, for the curious reader? :) – OJFord Jan 26, 2015 at 16:34

3 Just the attempt deserves way more credit :-) This code compiles (gcc-4.9) but gives no output - a little more information, like a blog post, would be great.
– Alfred Bratterud Mar 11, 2015 at 8:45

3 @OllieFord I found a version of it on a pastebin page and repasted it here: coliru.stacked-crooked.com/a/de06f2f63f905b7e .
– Johannes Schaub - litb Jul 18, 2015 at 11:07

130

Example

```
#include <iostream>

template <int N> struct Factorial
{
    enum { val = Factorial<N-1>::val * N };
};

template<>
struct Factorial<0>
{
    enum { val = 1 };
};

int main()
{
    // Note this value is generated at compile time.
    // Also note that most compilers have a limit on t
available.
    std::cout << Factorial<4>::val << "\n";
}
```

That was a little fun but not very practical.

To answer the second part of the question:
Is this fact useful in practice?

Short Answer: Sort of.

Long Answer: Yes, but only if you are a template daemon.

To turn out good programming using template meta-programming that is really useful for others to use (ie a library) is really really tough (though do-able). To Help boost even has MPL aka (Meta Programming Library). But try debugging a compiler error in your template code and you will be in for a long hard ride.

But a good practical example of it being used for something useful:

Scott Meyers has been working extensions to the C++ language (I use the term loosely) using the templating facilities. You can read about his work here 'Enforcing Code Features'

edited Dec 13, 2008 at 19:27

answered Oct 9, 2008 at 22:28

Loki Astari
264k ● 86 ● 342 ● 571

37 Dang there went concepts (poof) – Loki Astari Aug 22, 2009 at 0:03

7 I only have a small issue with the provided example - it does not exploit the (full) Turing-completeness of C++'s template system. Factorial can be found also using Primitive recursive functions, which are not turing-complete – Dalibor Frivaldsky Oct 6, 2013 at 22:39

4 aand now we have concepts lite – nurettin Oct 23, 2013 at 6:29

1 Over in 2017, we're pushing concepts even further back. Here's hope for 2020. – Ivaylo Valchev Jul 1, 2017 at 21:58

7 @MarkKegel 12 years later :D – Victor Jan 29, 2020 at 18:20

19

My C++ is a bit rusty, so the may not be perfect, but it's close.

```
template <int N> struct Factorial
{
    enum { val = Factorial<N-1>::val * N };
};

template <> struct Factorial<0>
{
    enum { val = 1 };
}

const int num = Factorial<10>::val;    // num set to 1
```

The point is to demonstrate that the compiler is completely evaluating the recursive definition until it reaches an answer.

Share Improve this answer Follow

edited Aug 28, 2009 at 15:31

Luc Touraille
81.9k ● 16 ● 99 ● 139

answered Oct 9, 2008 at 21:01

James Curran
103k ● 37 ● 185 ● 262

15

To give a non-trivial example: https://github.com/phresnel/metatrace , a C++ compile time ray tracer.

Note that C++0x will add a non-template, compile-time, turing-complete facility in form of `constexpr`:

```
constexpr unsigned int fac (unsigned int u) {
        return (u<=1) ? (1) : (u*fac(u-1));
}
```

You can use `constexpr`-expression everywhere where you need compile time constants, but you can also call `constexpr`-functions with non-const parameters.

One cool thing is that this will finally enable compile time floating point math, though the standard explicitly states that compile time floating point arithmetics do not have to match runtime floating point arithmetics:

> ```
> bool f(){
>     char array[1+int(1+0.2-0.1-0.1)]; //Must be ev
>     int  size=1+int(1+0.2-0.1-0.1); //May be evalu
>     return sizeof(array)==size;
> }
> ```

> It is unspecified whether the value of f() will be true or false.

Share Improve this answer

Follow

edited Dec 14, 2021 at 15:08

pizzapants184

169 ● 3 ● 9

answered Feb 8, 2010 at 15:45

Sebastian Mach

39.1k ● 8 ● 96 ● 130

10

The factorial example actually does not show that templates are Turing complete, as much as it shows that they support Primitive Recursion. The easiest way to show that templates are turing complete is by the Church-Turing thesis, that is by implementing either a Turing machine (messy and a bit pointless) or the three rules (app, abs var) of the untyped lambda calculus. The latter is much simpler and far more interesting.

What is being discussed is an extremely useful feature when you understand that C++ templates allow pure functional programming at compile time, a formalism that is expressive, powerful and elegant but also very complicated to write if you have little experience. Also notice how many people find that just getting heavily templatized code can often require a big effort: this is exactly the case with (pure) functional languages, which make compiling harder but surprisingly yield code that does not require debugging.

Share Improve this answer

Follow

answered Apr 29, 2009 at 9:56

Giuseppe Maggiore

1 Hey, what three rules you refer to, I wonder, by "app, abs, var"? I presume the first two are function application and abstraction (lambda definition(?)) respectively. Is that so? And what is the third one? Something having to do with variables? – Wizek Sep 27, 2016 at 16:47

I personally think that it would be generally better to a language support Primitive Recursion in the compiler than have it be Turing Complete, since a compiler for a language that supports compile-time Primitive Recursion could guarantee that any build will either complete or fail, but one whose build process is Turing Complete cannot, except by artificially constraining the build so it isn't Turing Complete. – supercat Feb 21, 2019 at 21:01

i sincerely hope the second paragraph is a joke – TopchetoEU Sep 14 at 10:49

6

Well, here's a compile time Turing Machine implementation running a 4-state 2-symbol busy beaver

```
#include <iostream>

#pragma mark - Tape

constexpr int Blank = -1;

template<int... xs>
class Tape {
public:
    using type = Tape<xs...>;
    constexpr static int length = sizeof...(xs);
```

```
};

#pragma mark - Print

template<class T>
void print(T);

template<>
void print(Tape<>) {
    std::cout << std::endl;
}

template<int x, int... xs>
void print(Tape<x, xs...>) {
    if (x == Blank) {
        std::cout << "_ ";
    } else {
        std::cout << x << " ";
    }
    print(Tape<xs...>());
}

#pragma mark - Concatenate

template<class, class>
class Concatenate;

template<int... xs, int... ys>
class Concatenate<Tape<xs...>, Tape<ys...>> {
public:
    using type = Tape<xs..., ys...>;
};

#pragma mark - Invert

template<class>
class Invert;

template<>
class Invert<Tape<>> {
public:
    using type = Tape<>;
};
```

```
template<int x, int... xs>
class Invert<Tape<x, xs...>> {
public:
    using type = typename Concatenate<
        typename Invert<Tape<xs...>>::type,
        Tape<x>
    >::type;
};

#pragma mark - Read

template<int, class>
class Read;

template<int n, int x, int... xs>
class Read<n, Tape<x, xs...>> {
public:
    using type = typename std::conditional<
        (n == 0),
        std::integral_constant<int, x>,
        Read<n - 1, Tape<xs...>>
    >::type::type;
};

#pragma mark - N first and N last

template<int, class>
class NLast;

template<int n, int x, int... xs>
class NLast<n, Tape<x, xs...>> {
public:
    using type = typename std::conditional<
        (n == sizeof...(xs)),
        Tape<xs...>,
        NLast<n, Tape<xs...>>
    >::type::type;
};

template<int, class>
class NFirst;

template<int n, int... xs>
class NFirst<n, Tape<xs...>> {
```

```
public:
    using type = typename Invert<
        typename NLast<
            n, typename Invert<Tape<xs...>>::type
        >::type
    >::type;
};

#pragma mark - Write

template<int, int, class>
class Write;

template<int pos, int x, int... xs>
class Write<pos, x, Tape<xs...>> {
public:
    using type = typename Concatenate<
        typename Concatenate<
            typename NFirst<pos, Tape<xs...>>::type,
            Tape<x>
        >::type,
        typename NLast<(sizeof...(xs) - pos - 1), Tape
    >::type;
};

#pragma mark - Move

template<int, class>
class Hold;

template<int pos, int... xs>
class Hold<pos, Tape<xs...>> {
public:
    constexpr static int position = pos;
    using tape = Tape<xs...>;
};

template<int, class>
class Left;

template<int pos, int... xs>
class Left<pos, Tape<xs...>> {
public:
    constexpr static int position = typename std::cond
```

```
        (pos > 0),
        std::integral_constant<int, pos - 1>,
        std::integral_constant<int, 0>
    >::type();

    using tape = typename std::conditional<
        (pos > 0),
        Tape<xs...>,
        Tape<Blank, xs...>
    >::type;
};

template<int, class>
class Right;

template<int pos, int... xs>
class Right<pos, Tape<xs...>> {
public:
    constexpr static int position = pos + 1;

    using tape = typename std::conditional<
        (pos < sizeof...(xs) - 1),
        Tape<xs...>,
        Tape<xs..., Blank>
    >::type;
};

#pragma mark - States

template <int>
class Stop {
public:
    constexpr static int write = -1;
    template<int pos, class tape> using move = Hold<po
    template<int x> using next = Stop<x>;
};

#define ADD_STATE(_state_)      \
template<int>                   \
class _state_ { };

#define ADD_RULE(_state_, _read_, _write_, _move_, _ne
template<>
class _state_<_read_> {
```

```
public:
    constexpr static int write = _write_;
    template<int pos, class tape> using move = _move_<
    template<int x> using next = _next_<x>;
};

#pragma mark - Machine

template<template<int> class, int, class>
class Machine;

template<template<int> class State, int pos, int... xs
class Machine<State, pos, Tape<xs...>> {
    constexpr static int symbol = typename Read<pos, T
    using state = State<symbol>;

    template<int x>
    using nextState = typename State<symbol>::template

    using modifiedTape = typename Write<pos, state::wr
    using move = typename state::template move<pos, mo

    constexpr static int nextPos = move::position;
    using nextTape = typename move::tape;

public:
    using step = Machine<nextState, nextPos, nextTape>
};

#pragma mark - Run

template<class>
class Run;

template<template<int> class State, int pos, int... xs
class Run<Machine<State, pos, Tape<xs...>>> {
    using step = typename Machine<State, pos, Tape<xs.

public:
    using type = typename std::conditional<
        std::is_same<State<0>, Stop<0>>::value,
        Tape<xs...>,
        Run<step>
    >::type::type;
```

```
};

ADD_STATE(A);
ADD_STATE(B);
ADD_STATE(C);
ADD_STATE(D);

ADD_RULE(A, Blank, 1, Right, B);
ADD_RULE(A, 1, 1, Left, B);

ADD_RULE(B, Blank, 1, Left, A);
ADD_RULE(B, 1, Blank, Left, C);

ADD_RULE(C, Blank, 1, Right, Stop);
ADD_RULE(C, 1, 1, Left, D);

ADD_RULE(D, Blank, 1, Right, D);
ADD_RULE(D, 1, Blank, Right, A);

using tape = Tape<Blank>;
using machine = Machine<A, 0, tape>;
using result = Run<machine>::type;

int main() {
    print(result());
    return 0;
}
```

Ideone proof run: https://ideone.com/MvBU3Z

Explanation: http://victorkomarov.blogspot.ru/2016/03/compile-time-turing-machine.html

Github with more examples: https://github.com/fnz/CTTM

Share Improve this answer

Follow

answered Mar 20, 2016 at 11:40

Victor Komarov
339 ● 4 ● 6

I think it's called template meta-programming.

5

Share Improve this answer

Follow

answered Oct 9, 2008 at 21:01

Tom Ritter

101k ● 31 ● 142 ● 174

2 This is the useful side of it. The downside is that I doubt most people (and certainly not me) will ever actually understand even a small percentage of what's going on in most of that stuff. It's horribly unreadable, unmaintainable stuff. – Michael Burr Oct 9, 2008 at 21:07

3 That's the drawback of the whole C++ language, I think. It's becoming a monster... – Federico A. Ramponi Oct 9, 2008 at 21:22

C++0x is promising to make a lot of it easier (and in my experience, the biggest issue is the compilers that don't support it fully, which C++0x won't help). Concepts in particular look like they'll clear things up, like getting rid of a lot of the SFINAE stuff, which is hard to read. – coppro Oct 9, 2008 at 23:19

1 @MichaelBurr The C++ committee doesn't care about unreadable, unmaintainable stuff; they just *love* to add features. – Sapphire_Brick Apr 20, 2020 at 22:43

It's also fun to point out that it is a purely functional language albeit nearly impossible to debug. If you look at

2

James post you will see what I mean by it being functional. In general it's not the most useful feature of C++. It wasn't designed to do this. It's something that was discovered.

Share Improve this answer

Follow

edited May 23, 2017 at 12:02

Community Bot

1 • 1

answered Oct 9, 2008 at 21:35

Matt Price

45.3k • 9 • 39 • 44

2

It may be useful if you want to compute constants at compile time, at least in theory. Check out template metaprogramming.

Share Improve this answer

Follow

edited Oct 9, 2008 at 23:18

answered Oct 9, 2008 at 21:00

user14070

1

An example which is reasonably useful is a ratio class. There are a few variants floating around. Catching the D==0 case is fairly simple with partial overloads. The real computing is in calculating the GCD of N and D and compile time. This is essential when you're using these ratios in compile-time calculations.

Example: When you're calculating centimeters(5)*kilometers(5), at compile time you'll be multiplying ratio<1,100> and ratio<1000,1>. To prevent overflow, you want a ratio<10,1> instead of a ratio<1000,100>.

Share Improve this answer

Follow

answered Oct 10, 2008 at 14:29

MSalters

179k ● 11 ● 164 ● 368

0

A Turing machine is Turing-complete, but that doesn't mean you should want to use one for production code.

Trying to do anything non-trivial with templates is in my experience messy, ugly and pointless. You have no way to "debug" your "code", compile-time error messages will be cryptic and usually in the most unlikely places, and you can achieve the same performance benefits in different ways. (Hint: 4! = 24). Worse, your code is incomprehensible to the average C++ programmer, and will be likely be non-portable due to wide ranging levels of support within current compilers.

Templates are great for generic code generation (container classes, class wrappers, mix-ins), but no - in my opinion the Turing Completeness of templates is **NOT USEFUL** in practice.

Share Improve this answer

Follow

answered Oct 10, 2008 at 13:24

Roddy

67.9k ● 44 ● 170 ● 280

4! may be 24, but what’s MY_FAVORITE_MACRO_VALUE! ? OK, I don't actually think this is a good idea, either.
– Jeffrey L Whitledge Oct 10, 2008 at 14:38

Just another example of how not to program :

```
template<int Depth, int A, typename B>
struct K17 {
    static const int x =
    K17 <Depth+1, 0, K17<Depth,A,B> >::x
    + K17 <Depth+1, 1, K17<Depth,A,B> >::x
    + K17 <Depth+1, 2, K17<Depth,A,B> >::x
    + K17 <Depth+1, 3, K17<Depth,A,B> >::x
    + K17 <Depth+1, 4, K17<Depth,A,B> >::x;
};
template <int A, typename B>
struct K17 <16,A,B> { static const int x = 1; };
static const int z = K17 <0,0,int>::x;
void main(void) { }
```

Post at [C++ templates are turing complete]

Share Improve this answer

Follow

answered Oct 22, 2009 at 13:26

lsalamon

8,128 ● 6 ● 52 ● 65

for the curious, the answer for x is pow(5,17-depth); – flownt Jul 6, 2010 at 13:38

1 Which is much simpler to see when you realize that template arguments A and B do nothing and delete them, and then replace all of the addition with `K17<Depth+1>::x * 5`. – David Stone Oct 5, 2014 at 1:44