`float` template parameter

A workaround

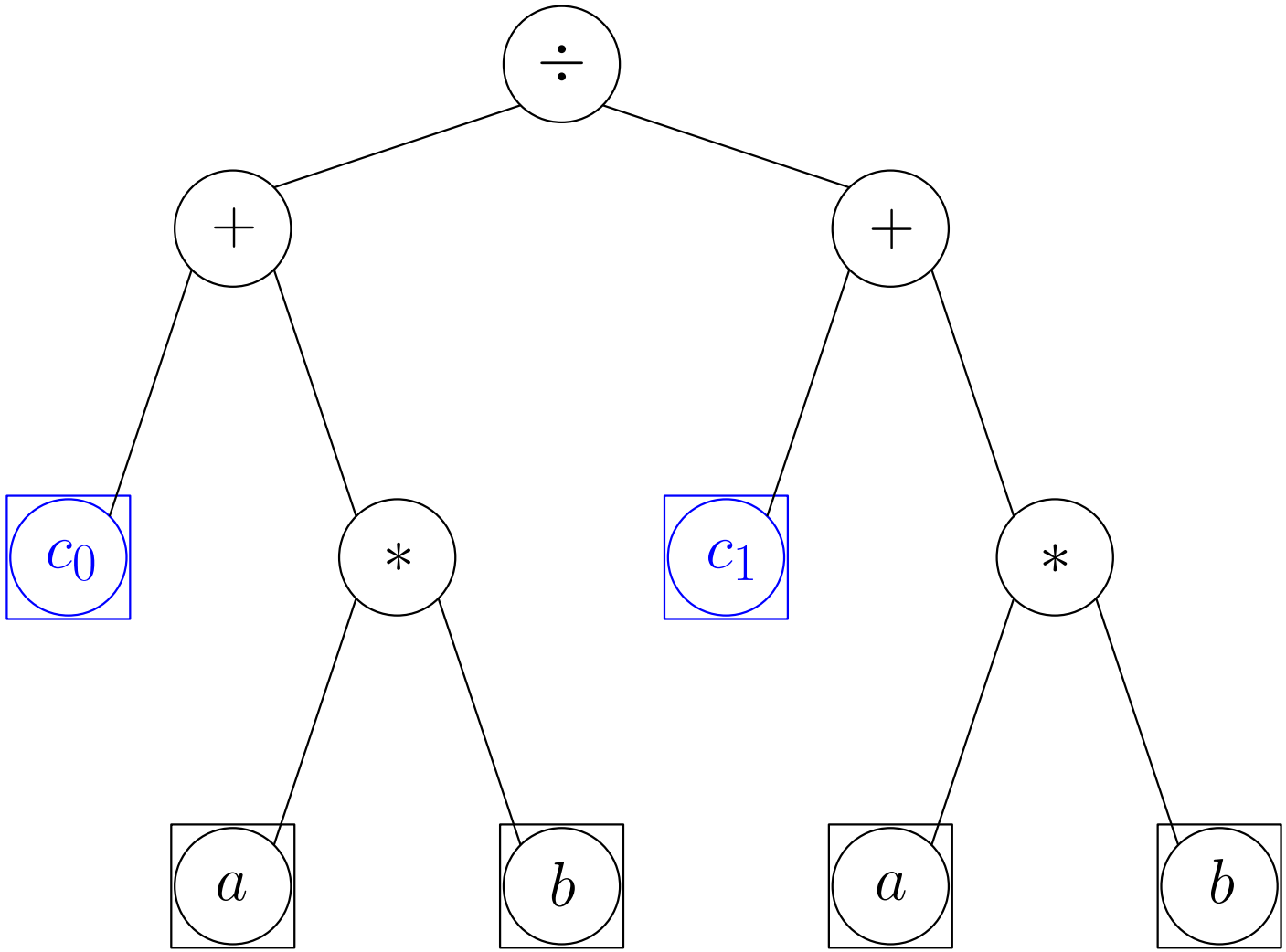
Dominic Jones

`github.com/DominicJones`

November 2018

From a previous talk...

Pruning at compile-time



Values as types

```
// C++ does not permit ‘auto’ to resolve as ‘float’
template<auto V>
struct _float
{
  constexpr operator auto() const { return V; }
};
```

```
// type distinguished by value
auto constexpr c0 = -4.2_f;
static_assert(c0 == _float<-4.2>{});
```

Identifiers as types

```
 123456789 123456789 12345
          10          20
1
2   // main.cpp
3   auto c1 = rand();
4   auto constexpr loc = &c1;
```

```
// & works something like this...
&c1 := hash(row, column, filename)
    := hash(3, 8, 'main.cpp')
```

```
// type distinguished by location
template<typename L, typename R>
auto operator+(L const &l, R const &r)
-> Binary<Add, L, R, &l, &r>
{
  return {l, r};
}
```

In another world...

D programming language

```d
// exactly what is wanted
struct _float(float v)
{
  static immutable auto value = v;
}
```

```d
// f-p constant folding is done in real or greater precision.
// Follows IEEE 754 rules and round-to-nearest is used.
static assert(_float!(9.0 / 3).value ==
              _float!(6.0 / 2).value);
```

```d
// Marks overflows as '_float!(infF)'
static assert(_float!(9.0 / 0).value ==
              _float!(6.0 / 0).value);
```

Back to C++

Why cannot this be done in C++?

- *I don't really know*
- "Two floats that are logically identical might not result in the same bit pattern, thus you would generate different templates."
- "The problem is that the textual representations of floating point values (constexpr) do not necessarily have the same value on different systems. And two different textual representations may map to the same value on some platforms, and different values on others."
- "NaN values have the curious property that they compare as 'unordered' with all values, even with themselves."

Placing some limitations

```
// disallow explicit instantiation
template struct _float<-4.2>;          // definition
extern template struct _float<-4.2>;  // declaration
```

```
// disallow Nan, Inf
auto v = _float<-4.2 / 0>{};
```

Is there a workaround?

- Yes,
- if you have compile-time time for it
- if compile-time programming is your favourite pastime

A rough sketch

"_float" type

```
// exponent ignored...
template<auto H, auto L, auto E>
struct _float
{
  auto constexpr static value =
    (H + float(L) / multiplier<10, E, 1>::value);

  constexpr operator auto() const { return value; }
};
```

```
// and for operator+
template<auto H, auto L, auto E>
auto constexpr operator-(_float<H, L, E>)
{
  return _float<(-H), (-L), E>{};
}
```

made palatable

```
// makes life easier
template<char...> struct mp_chars {};
```

```
// user-defined literal
template<char... Cs>
auto constexpr operator""_f()
{
  return make_float_t<0, 0, 0, 0,
                      sizeof...(Cs), mp_chars<Cs...> >{};
}
```

```
// seamless conversion to literals
auto constexpr v = -4.2_f;
float w = 2 * v;
```

Parsing

```
123.45_f

// represented as
mp_chars<'1', '2', '3', '.', '4', '5'>

H = '1', '2', '3'  // high chars
L = '4', '5'       // low chars
E = 4              // decimal offset
N = 6              // length
```

Terminal

```
// all chars consumed
template<auto H, auto L, auto E,
         auto I, auto N,
         template<char...> class CL>
struct make_float<H, L, E, I, N, CL<> >
{
  using type = _float<H, L, (N - E)>;
};
```

Decimal offset

```
// specialise for decimal character
template<auto H, auto L, auto E,
         auto I, auto N,
         template<char...> class CL, char... Cs>
struct make_float<H, L, E, I, N, CL<'.', Cs...> >
{
  auto static constexpr _E = I + 1;
  using type = make_float_t<H, L, _E, (I+1), N, CL<Cs...> >;
};
```

Digits

```
// distinguish high from low by decimal offset
template<auto H, auto L, auto E,
         auto I, auto N,
         template<char,char...> class CL, char C, char... Cs>
struct make_float<H, L, E, I, N, CL<C, Cs...> >
{
  auto static constexpr _D = (C >= '0' && C <= '9');
  auto static constexpr _H = (_D && E == 0)? 10*H+(C-'0'): H;
  auto static constexpr _L = (_D && E  > 0)? 10*L+(C-'0'): L;

  using type = make_float_t<_H, _L, E, (I+1), N, CL<Cs...> >;
};
```

`if constexpr`?

- Tried it initially
- Works too well! Either perfect or uncompilable
- Far easier to iteratively get something working
- Perhaps rewrite once parser is completed

Terminal (v2)

```
// still have a terminal function
template<auto H, auto L, auto E,
         auto I, auto N,
         template<char...> class CL>
auto constexpr make_float_fn(CL<> cl)
{
  return _float<H, L, (N - E)>{};
}
```

Decimal offset and digits (v2)

```
// return type deduced...
template<auto H, auto L, auto E,
         auto I, auto N,
         template<char,char...> class CL, char C, char... Cs>
auto constexpr make_float_fn(CL<C, Cs...> cl)
{
  if constexpr (C == '.')
  {
    auto constexpr _E = I + 1;
    return make_float_fn<H, L, _E, (I+1), N>(CL<Cs...>{});
  }
  else
  {
    auto constexpr _D = (C >= '0' && C <= '9');
    auto constexpr _H = (_D && E == 0)? 10*H+(C-'0'): H;
    auto constexpr _L = (_D && E  > 0)? 10*L+(C-'0'): L;
    return make_float_fn<_H, L, E, (I+1), N>(CL<Cs...>{});
  }
}
```

Standard library function?

```
// c-t string to float
using fp = ct_stof<Cs...>;

auto constexpr s = fp::significand;
auto constexpr e = fp::expoonent;
```

`float` template parameter

A workaround

Dominic Jones

`github.com/DominicJones`

November 2018