Compile time functions

An introduction

Dominic Jones

github.com/DominicJones

August 2018

Compile time functions

1. Like learning another language
2. Takes *much* longer than writing its plain old equivalent
3. Worth taking time to *play around* with implementation variations
4. Designing on paper saves a lot of grief!

rayseyfarth.com/asm/

Introduction to 64 Bit Assembly Language Programming for Linux and OS X

Ray Seyfarth

Background

This book is the third version of an assembly language textbook targeting beginning assembly language programmers. It teaches using the yasm assembler and the gdb debugger, though their use is normally behind the scene. This version of the book introduces the user to an integrated development environment named "ebe" which makes learning assembly language easier and more fun.

- Intended audience
  - Beginning assembly programmers
  - Any programmers who want to learn 64 bit programming
- Expected experience
  - 1 year of C or C++ experience

Resources provided to support the book

- ebe IDE
- Source code - tar gzipped
- Source code - zip
- Source code - individual files
- PDF slides for classroom presentation
- Videos (not yet started)
- Corrections

An introduction

1. The basics
2. Refactoring expressions
3. Functions as arguments
4. Cost of compilation
5. Mapping indices

The basics

Zero or more

```
// no state, but instantiable
template<class... T> struct list {};
```

```
// a more useful form...
template<class T, T...> struct seq {};
```

```
// _the_ operation
template<class L> struct front;

template<template<class...> class L, class T1, class... T>
struct front<L<T1, T...> >
{
  using type = T1;
};
```

Simple C++11 metaprogramming

With variadic templates, parameter packs and template aliases

Peter Dimov, 26.05.2015

I was motivated to write this after I read Eric Niebler's thought-provoking Tiny Metaprogramming Library article. Thanks Eric.

C++11 changes the playing field

The wide acceptance of Boost.MPL made C++ metaprogramming seem a solved problem. Perhaps MPL wasn't ideal, but it was good enough to the point that there wasn't really a need to seek or produce alternatives.

C++11 changed the playing field. The addition of variadic templates with their associated parameter packs added a compile-time list of types structure directly into the language. Whereas before every metaprogramming library defined its own type list, and MPL defined several, in C++11, type lists are as easy as

```
// C++11
template<class... T> struct type_list {};
```

and there is hardly a reason to use anything else.

Template aliases are another game changer. Previously, "metafunctions", that is, templates that took one type and produced another, looked like

```
// C++03
template<class T> struct add_pointer { typedef T* type; };
```

and were used in the following manner:

Some not so obvious things

Incomplete types

```
// could be written as a class template...
template<class T> using add_pointer = T*;
```

```
// (to be revisited)
template<template<class... T> class F> struct bind;
```

```
// ‘add_pointer’ is unqualified
//  but still satisfies ‘bind’!
using incomplete_type = bind<add_pointer>;
```

Lazy construction

```
// nesting defers type construction
template<int N> struct lazy_seq
{
  using type = std::make_index_sequence<N>;
};
```

```
// requiring additional '::type' wrapper
template<bool B, int NT, int NF>
using make_seq = std::conditional_t<B, lazy_seq<NT>,
                                       lazy_seq<NF> >;
```

Recursion

```
template<class... L> struct append;
```

```
// join lists
template<template<class...> class L1, class... T1,
         template<class...> class L2, class... T2,
         class... L>
struct append<L1<T1...>, L2<T2...>, L...>
{
  using type = typename append<L1<T1..., T2...>, L...>::type;
};
```

```
// ... until only one is left
template<template<class...> class L, class... T>
struct append<L<T...> >
{
  using type = L<T...>;
};
```

Refactoring expressions

Having and eating cake

```
// looks nice
r = (A + B)[i];
```

```
// works well
r = A[i] + B[i];
```

Expressive C++: Expression Optimization

Posted 2 days ago by Eric Niebler, under Boost

This entry is part of a series, Expressive C++»

Welcome back to Expressive C++, a series of articles about Embedded Domain-Specific Languages and Boost.Proto, a library for implementing them in C++. In previous articles, we discussed EDSLs as a way to provide expressive and powerful library interfaces. Well and good, but some might dismiss the whole "domain-specific language" thing as overly-cute operator overloading wankery. So this time around I'm going to talk about something near and dear to every C++ programmers' hearts: performance. I'll show you how to use Proto to perform optimizations your compiler can't do, rewriting inefficient expressions and improving runtime performance.

Linear Algebra

The linear algebra domain concerns itself with vectors and matrices. In this article I'll talk about vectors because the operations over them are easy to understand. Vector addition is element-wise. See figure 1:

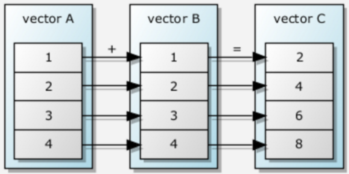


Figure 1: Vector addition

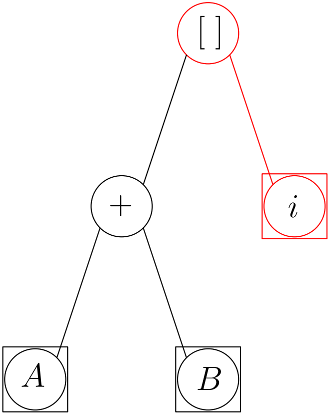
What does your C++ compiler know about linear algebra? Probably not much. For instance, given some vectors A and B, what would the compiler do with this?

```
int i = (A + B)[2];
```

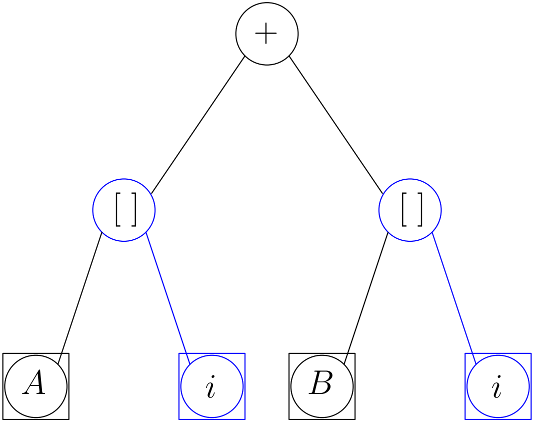
We humans who know the linear algebraic rule for vector addition know that the above is equivalent to:

```
int i = A[2] + B[2];
```

Having and eating cake



`(A + B)[i]`



`A[i] + B[i]`

Having and eating cake

```
// (A + B)[i]
Binary<Idx, Binary<Add, A, B>, int>
```

```
// apply special rules...
transform((A + B)[i])
```

```
// A[i] + B[i]
Binary<Add, Binary<Idx, A, int>,
            Binary<Idx, B, int> >
```

Having and eating cake

```
// default transform: do nothing
template<class E> auto transform(E e)
{
  return e;
}
```

Having and eating cake

```
// push 'Idx' operation down to the child nodes
template<class LF, class LL, class LR, class R>
auto transform(Binary<Idx, Binary<LF, LL, LR>, R> e)
{
  return
    transform(
      Binary<LF, Binary<Idx, decltype(transform(e.l.l)), R>,
                 Binary<Idx, decltype(transform(e.l.r)), R> >
        {{transform(e.l.l), transform(e.r)},
         {transform(e.l.r), transform(e.r)}});
}
```

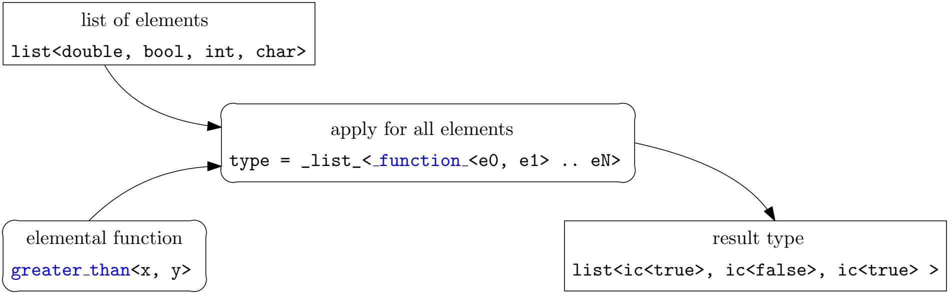
Having and eating cake

```cpp
// apply transforms to child nodes
template<class F, class L, class R>
auto transform(Binary<F, L, R> e)
{
  return
    Binary<F, decltype(transform(e.l)),
              decltype(transform(e.r))>
      {transform(e.l), transform(e.r)};
}
```

```cpp
// needed for cases like:
transform(((A+B) - (C+D))[i])
```

Functions as arguments

Splitting things up



Splitting things up

```
struct _1;  struct _2;
template<template<class...> class, class...> struct bind;
```

```
// apply
template<class L, class P> struct apply;

template<template<class...> class L, class T, class... Ts,
         template<class...> class F>
struct apply<L<T, Ts...>, bind<F, _1, _2> >
{
  using type = L<F<T, Ts>...>;
};
```

```
// greater_than
template<class X, class Y>
using greater_than = bool_t<(sizeof(X) > sizeof(Y))>;
```

```
// example
using L = list<int, bool, char, double>;
using R = apply_t<L, bind<greater_than, _1, _2> >;
```

Back to bind

```
// ‘unnamed’ variables of a function signature
struct _1;  struct _2;
```

```
// accepts a type of the form ‘F<T...>’
// and, optionally, some other types too
template<template<class...> class, class...> struct bind;
```

Leveraging incompleteness

```
// apply the transformation 'P' to list 'L'
template<class L, class P> struct apply;
```

```
// the magic step:
//   '_1' and '_2' serve only to
//   match the class specialisation
template<template<class...> class L, class T, class... Ts,
         template<class...> class F>
struct apply<L<T, Ts...>, bind<F, _1, _2> >
{
  using type = L<F<T, Ts>...>;
};
```

The rub

```
// 'greater_than' must be forwarded to an 'apply'
// specialised for two function variables
using R = apply_t<L, bind<greater_than, _1, _2> >;
```

Cost of compilation

Linear scaling

```
template<class S> struct concat;

template<int... I>
struct concat<seq<I...> >
{
  using type = seq<0, (1 + I)...>;
};
```

```
template<int N> struct make_seq;

template<> struct make_seq<0> { using type = seq<>; };

template<int N> struct make_seq
{
  using type = concat_t<make_seq_t<(N-1)> >;
};
```

```
using S = make_seq_t<5000>; // compile time: 3m 8s
```

Linear scaling

```
N = 4

  make_seq<3>
    make_seq<2>
      make_seq<1>
        make_seq<0> = seq<>

        concat = seq<0>  i.e.  seq<0, (1 + [null])...)>
      concat = seq<0,1>
    concat = seq<0,1,2>
  concat = seq<0,1,2,3>
```

Logarithmic scaling

```
template<class S1, class S2> struct concat;

template<int... I1, int... I2>
struct concat<seq<I1...>, seq<I2...> >
{
  using type = seq<I1..., (sizeof...(I1) + I2)...>;
};
```

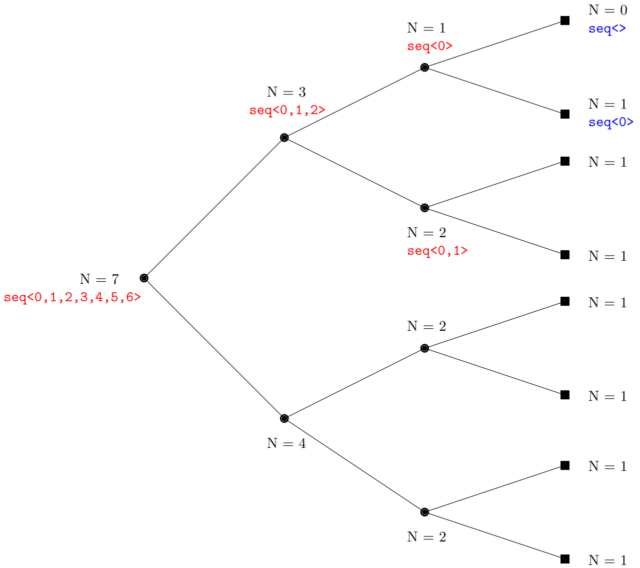
```
template<int N> struct make_seq;

template<> struct make_seq<0> { using type = seq<>; };
template<> struct make_seq<1> { using type = seq<0>; };

template<int N> struct make_seq
{
  using type = concat_t<make_seq_t<(N/2)>,
                        make_seq_t<(N-N/2)> >;
};
```

```
using S = make_seq_t<5000>; // compile time: 0.088s
```

Logarithmic scaling



Tinkering

```
template<int... I1, int... I2>
struct concat<seq<I1...>, seq<I2...> >
{
  static constexpr int _N1 = sizeof...(I1);
  using type = seq<I1..., (_N1 + I2)...>;
};
```

```
using S = make_seq_t<5000>; // compile time: 4.045s
```

More tinkering

```
template<int... I1, int... I2>
struct concat<seq<I1...>, seq<I2...> >
{
  using _N1 = std::integral_constant<int, sizeof...(I1)>;
  using type = seq<I1..., (_N1() + I2)...>;
};
```

```
using S = make_seq_t<5000>; // compile time: 0.148s
```

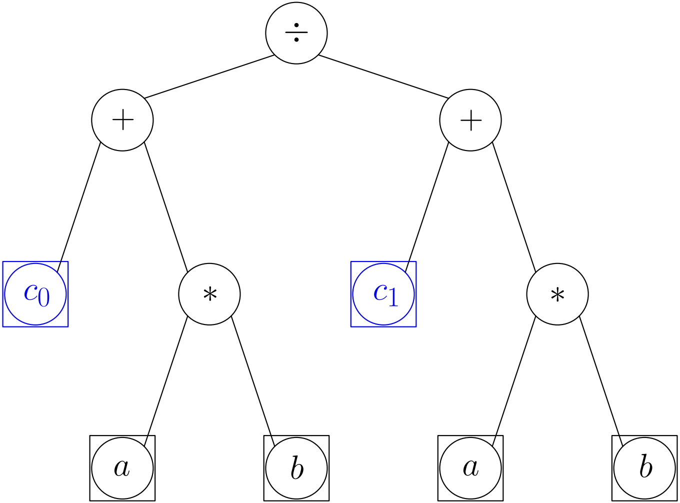
Mapping indices

Storing data off the tree

```
auto evaluate(A &a, B &b)
{
  // (to be revisited)
  auto c0 = 4.2_f;
  auto c1 = rand();

  // build expression tree
  auto t0 = a * b;
  auto t1 = c0 + t0;
  auto t2 = c1 + t0;
  return t1 / t2;
}
```

Storing data off the tree



Storing data off the tree

With operator overloading

```
Binary<Div,
  Binary<Add, C0,
    Binary<Mul, A, B> >
  Binary<Add, C1,
    Binary<Mul, A, B> > >
```

then flattening...

```
{Binary<Mul, A, B>,
 Binary<Add, C0, [...]>,
 Binary<Add, C1, [...]>,
 Binary<Div, [...], [...]>}
```

Mapping nodes to data

Look-up of `C0` and `C1` required

```
L = {A,                                   //  0
     B,                                   //  1
     Binary<Mul, A, B>,                   //  2
     Binary<Add, C0, [...]>,              //  3  <--
     Binary<Add, C1, [...]>,              //  4  <--
     Binary<Div, [...], [...]>}           //  5
```

```
array<float, 2> c = {4.2, rand()};
```

Indices of constants in ‘L’

```
IC = {3, 4}
```

Map elements in ‘L’ to storage offsets

```
DC = {2, 2, 2, 0, 1, 2}
```

Dual function

```
// homework
//-----------
DC_SIZE = 6      \
                  --- Dual --->     DC = {2, 2, 2, 0, 1, 2}
IC = {3, 4}      /                        0  1  2  3  4  5
```

```
// step 1: index the input
IC_MAP = {{3, 0}, {4, 1}}
```

```cpp
template<class L, class S, std::size_t I, std::size_t NI, std::size_t NJ>
struct mp_dual_impl;

template<template<class...> class L,
        template<class SX, SX...> class S, class ST, ST... SVs,
        std::size_t I, std::size_t NI, std::size_t NJ>
struct mp_dual_impl<L<>, S<ST, SVs...>, I, NI, NJ>
{
  using type = mp_list<>;
};

template<template<class...> class L,
        template<class SX, SX...> class S, class ST, ST SV, ST... SVs,
        std::size_t I, std::size_t NI, std::size_t NJ>
struct mp_dual_impl<L<>, S<ST, SV, SVs...>, I, NI, NJ>
{
  using _head = std::integral_constant<std::size_t, NJ>;
  using _tail = typename mp_dual_impl<L<>, S<ST, SVs...>, I + 1, NI, NJ>::type;
  using type = mp_append<mp_list<_head>, _tail>;
};


template<class L, class S, std::size_t I, std::size_t NI, std::size_t NJ, std::size_t J>
struct mp_dual_impl_valid;

template<template<class...> class L, class T, class... Ts,
        template<class _SX, _SX...> class S, class ST, ST SV, ST... SVs,
        std::size_t I, std::size_t NI, std::size_t NJ, std::size_t J>
struct mp_dual_impl_valid<L<T, Ts...>, S<ST, SV, SVs...>, I, NI, NJ, J>
{
  auto static constexpr JD = (I == J)? mp_second<T>::value: NJ;
  using _head = std::integral_constant<std::size_t, JD>;

  using _tail = typename mp_if_c<bool(I == J),
                                mp_dual_impl<L<Ts...>, S<ST, SVs...>, I + 1, NI, NJ>,
                                mp_dual_impl<L<T, Ts...>, S<ST, SVs...>, I + 1, NI, NJ>
                                >::type;

  using type = mp_append<mp_list<_head>, _tail>;
};
```

```cpp
template<class L, class S, std::size_t I, std::size_t NI, std::size_t NJ>
struct mp_dual_impl_invalid;

template<template<class...> class L, class T, class... Ts,
        template<class _SX, _SX...> class S, class ST, ST SV, ST... SVs,
        std::size_t I, std::size_t NI, std::size_t NJ>
struct mp_dual_impl_invalid<L<T, Ts...>, S<ST, SV, SVs...>, I, NI, NJ>
{
  using _head = std::integral_constant<std::size_t, NJ>;
  using _tail = typename mp_dual_impl<L<Ts...>, S<ST, SVs...>, I + 1, NI, NJ>::type;
  using type = mp_append<mp_list<_head>, _tail>;
};


template<template<class...> class L, class T, class... Ts,
        template<class _SX, _SX...> class S, class ST, ST SV, ST... SVs,
        std::size_t I, std::size_t NI, std::size_t NJ>
struct mp_dual_impl<L<T, Ts...>, S<ST, SV, SVs...>, I, NI, NJ>
{
  auto static constexpr J = mp_first<T>::value;

  using type = typename mp_if_c<bool(J < NI),
    mp_dual_impl_valid<L<T, Ts...>, S<ST, SV, SVs...>, I, NI, NJ, J>,
    mp_dual_impl_invalid<L<T, Ts...>, S<ST, SV, SVs...>, I, NI, NJ>
    >::type;
};

template<typename L, std::size_t NI>
struct mp_dual_entry
{
  using IS = std::make_index_sequence<NI>;
  auto static constexpr NJ = mp_size<L>::value;
  using M = mp_reversed_map_from_list<L>;
  using MS = brigand::sort<M, brigand::bind<mp_map_incr, brigand::_1, brigand::_2> >;
  using D = mp_dual_impl<MS, IS, 0, NI, NJ>;
  using type = typename D::type;
};

template<class L, std::size_t NI> using mp_dual = typename mp_dual_entry<L, NI>::type;
```

github.com/DominicJones/snippets/blob/master/Cxx/mp_functions.cpp

Four last things

Mutually exclusive?

```
auto constexpr list = [](auto ...x)
{
  return [=](auto f){ return f(x...); };
};
```

```
auto constexpr length = [](auto x)
{
  return x([](auto ...f){ return sizeof...(f); });
};
```

```
auto constexpr n = length(list(true, 'c', 3.14));
```

g++ demangle

```
#include <cxxabi.h>

struct Demangle
{
  static std::string eval(char const * name)
  {
    int status(-4);
    char * realname;
    realname = abi::__cxa_demangle(name, 0, 0, &status);
    std::string result(realname);
    free(realname);
    return result;
  }
};

template<typename Expr_t>
std::string demangle(Expr_t const &expr)
{
  return Demangle::eval(typeid(expr).name());
}

template<typename Expr_t>
std::string demangle()
{
  return Demangle::eval(typeid(Expr_t).name());
}
```

```
// very helpful
std::cout << demangle(complicated_type{}) << std::endl;
```

Low hanging fruit

```
// permit ‘auto’ to resolve as ‘float’
template<auto V>
struct _float
{
  constexpr operator auto() const { return V; }
};

// only interested in the type
auto constexpr c0 = 4.2_f;
static_assert(c0 == _float<4.2>{});
```

D has had this for years...

dlang.org/spec/template.html

The third realm

```
 123456789 123456789 12345
        10        20
1
2  // main.cpp
3  auto c1 = rand();
4  auto constexpr c1_loc = &c1;
```

```
// reflect location
&c1 := hash(row, column, filename)
    := hash(3, 8, `$PATH/main.cpp`)
```

A rough draft on it...

`github.com/DominicJones/articles/blob/master/cxx-sg7-varid`

Compile time functions

An introduction

Dominic Jones

github.com/DominicJones

August 2018