Compile-time factorization

[crazy template meta-programming]

Vladimir Mirnyy

C++ Meetup, 15 Sep. 2015, C Base, Berlin

The origin of this task

GFFT - Generative Fast Fourier Transforms (GFFT) gfft.sf.net

- Key feature: Length of the transform N is static constant
- Common and the most efficient case $N = 2^P$
- How to implement a general size N transform?

Factorize N at compile-time!

Factorization of the natural number N is its decomposition into a set of prime factors, the product of which equals N.

Implementation is feasible, functionality is unknown

The algorithm: improved trial division

- Worst case: Try all $m = 2, 3, ..., m^2 \leq N$
- Try factor 2 and then the factors in the form $m = 2k + 1$

$$k = 1, 2, ... \qquad m = 3, 5, ...$$

- Try 3 and 5, then use $m = 2 \cdot 3k + r, \quad r = 1, 2, 3, 4, 5$

$$k = 1, 2, ... \qquad m = 7, 11, 13, 17, ...$$

- Next formula: $m = 2 \cdot 3 \cdot 5k + r = 30k + r, \quad k = 1, 2, ...$
  Remainders: $r > 0 \quad \&\& \quad r < 30 \quad \&\& \quad r\%\{2, 3, 5\} \neq 0$

$$r = 1, 7, 11, 13, 17, 19, 23, 29$$

$$k = 1, ... \qquad m = 31, 37, 41, 43, 47, 49, 53, 59, ...$$

The algorithm summary

1. Test the initial primes less than q
2. Start the trial division using factors in the form $m = q \cdot k + r$, $k = 1, 2, ...$, iterating over the set of remainders r for each k
3. Reduce $N := N/m$, if $N\%m == 0$

Formula $q \cdot k + r$	**Eliminated factors**	k **range to the next**	**Number of remainders**	**% of** $\mathbb{N}$
$2k + 1$	2	1,2	1	50%
$6k + r$	2,3	1,2,3,4	2	33%
$30k + r$	2,3,5	1,2,...,6	8	27%
$210k + r$	2,3,5,7	1,2,...,10	48	23%
$2310k + r$	2,3,5,7,11	1,2,...,12	484	21%
...				

- Sieve of Eratosthenes in a functional way
- Each formula reduces the number of trial divisions
- But the set of remainders r grows very quickly

Implementation using variadic templates

- `typelist` - compile-time array to store the remainders and the result of factorization - empty class template with variadic template parameters
- `typelist_cat` - concatenation of two typelists or insertion in the beginning or in the end

```
template<typename ...List> struct typelist;

template <typename List1, typename List2> struct typelist_cat;

template <typename ...List1, typename ...List2>
struct typelist_cat<typelist<List1...>, typelist<List2...>>
{
  using type = typelist<List1..., List2...>;
};
template <typename T, typename ...List2>
struct typelist_cat<T, typelist<List2...>>
{
  using type = typelist<T, List2...>;
};
template <typename T, typename ...List1>
struct typelist_cat<typelist<List1...>,T>
{
  using type = typelist<List1... ,T>;
};
```

Implementation: sint and spair

- `sint` - holder for static integer constant
- `spair` - holder for a pair of types
- Result of factorization will be stored as `typelist` of pairs (prime factor, its power) with non-zero powers only

Define static int and pair holders:

```
typedef unsigned long int_t;

template<int_t N>
struct sint
{
  static const int_t value = N;
};

template<typename T1, typename T2>
struct spair
{
  typedef T1 first;
  typedef T2 second;
};
```

Implementation: factorization class template

1. Goes through `InitialPrimesList`
   (All entries in `InitialPrimesList` are less than Q)

```
template<typename N,
unsigned int Q = StartQ,
typename StartList = InitialPrimesList>
struct factorization;

// Factorization using trial division from InitialPrimesList
template<int_t N, unsigned int Q, typename H, typename ...Tail>
struct factorization<sint<N>, Q, typelist<H,Tail...> >
{
  static const int_t candidate = H::value;
  using trial = try_factor<N, candidate>;
  static const int_t P = trial::power;
  using T = spair<sint<candidate>, sint<P> >;
  using nextN = sint<N/trial::factor>;
  using next = typename factorization<nextN,Q,typelist<Tail...>>::type;
  using type = typename std::conditional<(P > 0),
     typename typelist_cat<T, next>::type, next>::type;
};
```

Implementation: factorization class template

2. Starts `factor_loop`, when trial division based on `InitialPrimesList` is completed
3. Contains exit conditions for the case, if N is fully factored by `InitialPrimesList`

```
template<int_t N, unsigned int Q>
struct factorization<sint<N>, Q, typelist<> >
{
  static const int_t candidate = Q + 1;
  using type = typename factor_loop<N,Q,1,Remainders,(candidate*candidate > N)>::type;
};

// End of factorization
template<unsigned int Q, typename H, typename ...List>
struct factorization<sint<1>, Q, typelist<H, List...>>
{
  using type = typelist<>;
};
template<unsigned int Q>
struct factorization<sint<1>, Q, typelist<>>
{
  using type = typelist<>;
};
```

Implementation: `factor_loop`

Iterates over both k and remainders

```
template<int_t N, unsigned int Q, int_t K, typename RList, bool doExit = false>
struct factor_loop;

// Loop over remainders
template<int_t N, unsigned int Q, int_t K, typename H, typename ...Tail>
struct factor_loop<N,Q,K,typelist<H,Tail...>,false>
{
  static const int_t candidate = Q*K + H::value;
  using trial = try_factor<N, candidate>;
  using T = spair<sint<candidate>, sint<trial::power> >;
  using nextIter = typename factor_loop<N/trial::factor, Q, K, typelist<Tail...>,
                          (candidate*candidate > N)>::type;
  using type = typename std::conditional<(trial::power > 0),
              typename typelist_cat<T,nextIter>::type, nextIter>::type;
};

// Increment K
template<int_t N, unsigned int Q, int_t K>
struct factor_loop<N,Q,K,typelist<>,false>
{
  static const int_t candidate = Q*(K+1) + 1;
  using type = typename factor_loop<N,Q,K+1,Remainders,(candidate*candidate > N)>::type;
};
```

Implementation: `factor_loop` exit conditions ($m^2 > N$)

1. $N > 1$, then the current N is the last prime factor of the original input N
2. $N == 1$, then N is factored, return empty typelist

```
// N > 1
template<int_t N, unsigned int Q, int_t K, typename RList>
struct factor_loop<N,Q,K,RList,true>
: public factor_loop<N,Q,K,typelist<>,true> {};

template<int_t N, unsigned int Q, int_t K>
struct factor_loop<N,Q,K,typelist<>,true>
{
  using type = typelist<spair<sint<N>, sint<1>>>;
};

// N == 1
template<unsigned int Q, int_t K, typename RList>
struct factor_loop<1,Q,K,RList,true>
: public factor_loop<1,Q,K,typelist<>,true> {};

template<unsigned int Q, int_t K>
struct factor_loop<1,Q,K,typelist<>,true>
{
  using type = typelist<>;
};
```

Implementation: `try_factor`

For the given N and the trial factor m calculate the power p and the full factor $f = m^p$ so that N is divisible by f

```
template<int_t N, int_t TrialFactor, bool C = (N % TrialFactor == 0)>
struct try_factor;

template<int_t N, int_t TrialFactor>
struct try_factor<N, TrialFactor, true> {
  typedef try_factor<N/TrialFactor, TrialFactor> next;
  static const int_t power = next::power + 1;
  static const int_t factor = next::factor * TrialFactor;
};

template<int_t N, int_t TrialFactor>
struct try_factor<N, TrialFactor, false> {
  static const int_t power = 0;
  static const int_t factor = 1;
};

template<int_t TrialFactor>
struct try_factor<0, TrialFactor, true> {
  static const int_t power = 0;
  static const int_t factor = 1;
};
```

Implementation: two constexpr functions instead of `try_factor`

```
constexpr int_t factor_power(int_t N, int_t TrialFactor)
{
  return (N % TrialFactor == 0) ? 1+factor_power(N/TrialFactor, TrialFactor) : 0;
}

constexpr int_t full_factor(int_t N, int_t TrialFactor)
{
  return (N % TrialFactor == 0) ? TrialFactor*full_factor(N/TrialFactor, TrialFactor) : 1;
}
```

Implementation: printing out the results

```
template<typename List> struct typelist_out;

template<typename T, typename ...Args>
struct typelist_out<typelist<T, Args...> >
{
  static void print(std::ostream& os = std::cout, const char sep = '\t')
  {
    os << T::value << sep;
    typelist_out<typelist<Args...>>::print(os, sep);
  }
};

template<typename T1, typename T2, typename ...Args>
struct typelist_out<typelist<spair<T1,T2>, Args...> >
{
  static void print(std::ostream& os = std::cout, const char sep = '\t')
  {
    os << T1::value << "^" << T2::value << sep;
    typelist_out<typelist<Args...>>::print(os, sep);
  }
};

template<>
struct typelist_out<typelist<>>
{
  static void print(std::ostream& os = std::cout, const char sep = '\t')
  {
    os << std::endl;
  }
};
```

Implementation: `InitialPrimesList` and `Remainders`

Example for the formula $6k + r$

```
static const int_t StartQ = 2*3;

using InitialPrimesList = typelist<sint<2>,sint<3>,sint<5>>;

using Remainders = typelist<sint<1>,sint<5>>;


static const int_t N = 12345;

int main(int argc, char *argv[])
{
    std::cout << "Compile-time factorization of " << N << ": ";
    typelist_out<factorization<sint<N> >::type>::print();
    return 0;
}
```

Benchmarks: Test cases and compilers

Test cases:

1. 65521 - the largest prime of type `unsigned short`
2. 4294967291 - the largest prime of type `unsigned int`
3. $18446744073709551615 = 3 \cdot 5 \cdot 17 \cdot 257 \cdot 641 \cdot 65537 \cdot 6700417$ = `std::numeric_limits<unsigned long>::max()`

Tested compilers: g++ 4.8.3, clang 3.4.2, Intel C++ 14.0.2, MSVC 12

Some cases to make your compiler (and computer) crying:

1. $9223372036854775807 = 7^2 \cdot 73 \cdot 127 \cdot 337 \cdot 92737 \cdot 649657$ still compiles taking around 4 seconds
2. $9223372036854775806 = 2 \cdot 3 \cdot 715827883 \cdot 2147483647$ - does not compile
3. 18446744073709497 - another big prime

Benchmarks: gcc 4.8.3

- Compiles all the test cases
- Tries to compile more complicated cases using all available RAM
- The third formula $210k + r$ is the fastest

`g++ -O3 -std=c++11 -ftemplate-depth-1000000`

g++ 4.8.3	**6k+r**	**30k+r**	**210k+r**	**2310k+r**
65521	0.4	0.4	0.4	9.1
4294967291	10.0	3.5	3.2	80.2
18446744073709551615	10.0	3.4	3.3	81.8

Compile-time in seconds

Benchmarks: clang 3.4.2

- The fastest one of all tested compilers
- Crashes after a few seconds on more complicated cases
- The third formula $210k + r$ is the fastest

`clang -O3 -std=c++11 -ftemplate-depth-1000000`

clang 3.4.2	**6k+r**	**30k+r**	**210k+r**	**2310k+r**
65521	0.5	0.4	0.4	5.1
4294967291	-	-	2.2	10.4
18446744073709551615	-	-	2.2	10.4

Compile-time in seconds

Benchmarks: Intel C++ 14.0.2

- The slowest one under Linux
- Crashes on more complicated cases
- The third formula $210k + r$ is the fastest

`icpc -O3 -std=c++11 -ftemplate-depth-1000000`

Intel C++ 14.0.2	**6k+r**	**30k+r**	**210k+r**	**2310k+r**
65521	0.6	0.5	0.6	111.5
4294967291	-	-	6.6	305.2
18446744073709551615	-	-	6.5	306.4

Compile-time in seconds

Benchmarks: Visual Studio 2013 (MSVC 12)

- Template depth is fixed to 2048 and cannot be changed
- Compiles less test cases than other tested compilers
- The third formula $210k + r$ is the only solution for all test cases

cl.exe	**6k+r**	**30k+r**	**210k+r**	**2310k+r**
65521	0.4	0.7	0.8	-
4294967291	-	-	43.2	-
18446744073709551615	-	-	42.7	-

Compile-time in seconds

Conclusions

- A reliable prime factorization at compile-time for all 4-byte integers using template meta-programming with variadic template parameters
- The optimal generator formula for the trial division is $210k + r$
- The article and the link to the source code are available under

  blog.scientificcpp.com or blog.scpp.eu
- The source code contains additionally:
  - The original implementation using Loki::Typelist instead of `typelist<...>` for non-C++11 compilers
  - Generation of remainders and prime numbers at compile-time using the same generator formulas for both candidate generation and primality testing

Thanks a lot for your attention!