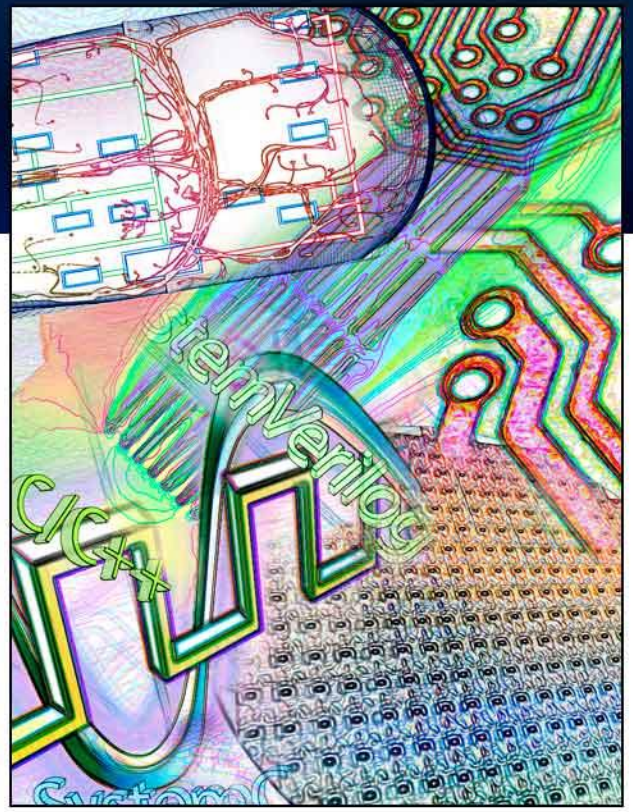


C++ Metaprogramming
Journey from simple to insane and back

Fedor G Pikus

Mentor Graphics, Design2Silicon Division

CPPCon, September 2015

Mentor Graphics®

This talk is

- about C++ template metaprogramming
- not particularly advanced (sorry, may be next time?)
- about practical metaprogramming in everyday applications
  - will show basic guidelines for "real-life" programming
  - template library developers already know and do all of it
  - this is not hard; it's merely useful
- example-driven
- accurate (or strives to be) but not precise
- the one where questions are encouraged

What is Metaprogramming?

- Metaprogramming is the writing of computer programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at compile time that would otherwise be done at run-time.
- Practically: using the C++ template system to perform computation at compile-time.

Why use metaprogramming?

- Move computations from run-time to compile-time
- Code generation, using compile-time computations
  - Reduce code duplication
- Enable "self-adapting" code (portability, efficiency)
  - 32-bit vs 64-bit
  - Eliminate deep run-time conditionals or indirections
  - Sort using random-access vs bidirectional iterators
- Facilitate domain-specific programming
  - Create expressive languages for solving specific problems
  - Scientific programming with units using (fairly) natural notation
  - Programming state machines
- Express complex software patterns and concepts in a more straightforward way

Tools of metaprogramming

- Template:

```
template <typename T> struct SizeOfT {
  enum { value = sizeof(T) };
};
```

- In C++11 can also use static constexpr data members:
  - In C++03 static constants (usually) must be defined somewhere

```
template <typename T> struct SizeOfT {
  static constexpr value = sizeof(T);
};
```

- Template metafunction Sizeof<T>::value
  - C++11 convention: numeric results are named "value", type results are named "type" (improves reusability)

Tools of metaprogramming

- Template:

```
template <typename T> struct SizeOfT {
  enum { value = sizeof(T) };
};
template <typename T> struct TPtr {
  typedef T* type;
};
```

- Template metafunction Sizeof<T>::value
  - C++11 convention: numeric results are named “value”, type results are named “type”

Tools of metaprogramming

- Template arguments can be types or numbers
  - or templates, but we're not going to need that here

```
template <size_t N> struct IsEven {
  enum { value = (N % 2) ? 0 : 1 };
};
```

Tools of metaprogramming

- Template specializations:

```
template <typename T> struct IsChar {
  enum { value = 0 };        // Works for most types
};
template <> struct IsChar<char> {
  enum {value = 1 };
};
```

Tools of metaprogramming

- Template specializations (you can have more than one):

```
template <typename T> struct IsChar {
  enum { value = 0 };        // Works for most types
};
template <> struct IsChar<char> {
  enum { value = 1 };
};
template <> struct IsChar<unsigned char> {
  enum { value = 1 };
};
```

- Forgetting anything?

Tools of metaprogramming

- Template specializations (you can have more than one):

```
template <typename T> struct IsChar {
  enum { value = 0 };        // Works for most types
};
template <> struct IsChar<char> {
  enum { value = 1 };
};
template <> struct IsChar<const char> {
  enum { value = 1 };
};
```

- And volatile, and const unsigned, volatile signed, etc…

Tools of metaprogramming

- Partial template specializations:

```
template <typename T> struct NoCV {
  typedef T type;
};
template <typename T> struct NoCV<const T> {
  typedef T type;
};
```

… also volatile and const volatile …

- C++11 has remove_cv<> metafunction!

```
template <typename T> struct IsAnyChar {
  enum { value =
    IsChar<typename NoCV<T>::type>:: value };
};
```

Note on C++11 syntax

- C++11 adds type aliases that save us from typing ::value everywhere

```
template <typename T> struct NoCV {
  using type = T;
};
template <typename T> struct NoCV<const T> {
  using type = T;
};
template <typename T> struct IsAnyChar {
  enum { value = IsChar<typename NoCV<T>>::value };
};
```

- Metafunction definitions are no simpler, calls are simpler

Not Tools of metaprogramming

- Variables
- for-loops

Tools of metaprogramming

- Recursion
- Termination is usually done via specialization
- Sum(N) = N + (N-1) + (N-2) + … + 1

```
template <size_t N> struct Sum {
  enum { value = N + Sum<N-1>::value };
};

template <> struct Sum<1> {
  enum { value = 1 };
};
```

- Compilers implement this reasonably fast but have recursion depth limits (constexpr functions often have higher limits and are much faster)

Types of metaprograms

- Compile-time numeric calculations
  - No run-time code is generated
  - C++11 constexpr functions can do this too
  - Classic example – computing factorial n!

New example – computing log2(x) for x=2^N

- Power of 2 numbers have exactly one bit set
  - These numbers frequently occur as sizes of types
- log2(x) counts the position of the set bit
  - 0001→0, 0100→2, etc
- Implementation: shift right by 1 until you get 1, count how many shifts were needed
- But – no loops, remember?

New example – computing log2(x) for x=2^N

- Power of 2 numbers have exactly one bit set
  - These numbers frequently occur as sizes of types
- log2(x) counts the position of the set bit
  - 0001→0, 0100→2, etc
- Recursive implementation: shift (count 1) and add log2() of the shifted value (N/2)

```
template <size_t N> struct Log2 {
  enum { value = 1 + Log2<N/2>::value };
};
template <> struct Log2<1> {
  enum { value = 0 };
};
```

You would not do this to your program, why do it to your compiler?

- This is a very slow way to compute log2()
- It may not matter if it's not done often at compilation
- We can do it faster:

```
log2(N) = (((N & 0xFFFFFFFF00000000UL) != 0) << 5) |
          (((N & 0xFFFF0000FFFF0000UL) != 0) << 4) |
          (((N & 0xFF00FF00FF00FF00UL) != 0) << 3) |
          (((N & 0xF0F0F0F0F0F0F0F0UL) != 0) << 2) |
          (((N & 0xCCCCCCCCCCCCCCCCUL) != 0) << 1) |
          ((N & 0xAAAAAAAAAAAAAAAAUL) != 0);
```

You would not do this to your program, why do it to your compiler?

- This is a very slow way to compute log2()
- It may not matter if it's not done often at compilation
- We can do it faster:

```
template <size_t N> struct Log2 {
enum { value =
        (((N & 0xFFFFFFFF00000000UL) != 0) << 5) |
        (((N & 0xFFFF0000FFFF0000UL) != 0) << 4) |
        (((N & 0xFF00FF00FF00FF00UL) != 0) << 3) |
        (((N & 0xF0F0F0F0F0F0F0F0UL) != 0) << 2) |
        (((N & 0xCCCCCCCCCCCCCCCCUL) != 0) << 1) |
        ((N & 0xAAAAAAAAAAAAAAAAUL) != 0) };
};
```

One caveat

- This only works on 64-bit systems
- No problem:

```
template <size_t N, size_t s> struct Log2Helper;
template <size_t N> struct Log2Helper<N, 8> { … };
template <size_t N> struct Log2Helper<N, 4> {
enum { value = (((N & 0xFFFF0000) != 0) << 4) |
        (((N & 0xFF00FF00) != 0) << 3) |
        (((N & 0xF0F0F0F0) != 0) << 2) |
        (((N & 0xCCCCCCCC) != 0) << 1) |
        ((N & 0xAAAAAAAA) != 0) };
};
template <size_t N> struct Log2 :
  public Log2Helper<N, sizeof(N)> {};
```

Tools of metaprogramming

- If-then-else:

```
template <bool N, typename T, typename F> struct IF {
  typedef T type;
};

template <typename T, typename F> struct IF<false,T,F> {
  typedef F type;
};
```

- C++11 has this very construct as std::conditional

Tools of metaprogramming

- If-then-else for numbers:

```
template <bool N, size_t T, size_t F> struct IF {
  enum { value = T };
};

template <size_t T, size_t F> struct IF<false,T,F> {
  enum { value = F };

template <size_t N> struct Log2 {
  enum { value = IF<sizeof(N) == 8, …, … >};
};
```

Compile-time (static) assertions

- What if N is not a power of 2?
- Code for Log2 should not compile

```
template <size_t N> struct IsPower2 {
  enum { value = (N & (N-1)) == 0 };
};

static_assert(IsPower2<N>::value, "N must be 2^n");
```

- No C++11? There are older ways to do compile assert

```
template <bool c> struct StaticAssert; // Undefined!
template <> struct StaticAssert<true> {};
```

- Error messages are non-obvious and depend on the compiler

Guidelines for using metaprogramming in everyday application programming

- Compile-time numeric calculations
  - Also consider constexpr functions in C++11
  - Use precomputed static values when it's simpler (π)

Types of metaprograms

- Compile-time numeric calculations
- Code generation using numeric calculations
  - The more common use of numeric calculations

Example – computing x^N

- The C way: pow(x, N)
  - operates on doubles, not very efficient for large integer N
- The metaprogramming way: $x^N = x * x^{N-1}$

```
template <size_t N> struct Pow {
  double operator()(double x) const
    { return x*Pow<N-1>()(x); }
};

template <> struct Pow<1> {
  double operator()(double x) const { return x; }
};

template <> struct Pow<0> {
  double operator()(double x) const { return 1; }
};
```

Example – computing x^N

- Test – computing x^N for random x, 2^{27} times
- pow(x, 16) – 9 seconds, Pow<16>()(x) – 0.5 seconds
- pow(x, 20) – 9 seconds, Pow<20>()(x) – 0.9 seconds
- x^{16} is better computed as $x^8 * x^8$

```
template <size_t N> struct Pow {
  double operator()(double x) const {
    return Pow<N/2>()(x)*Pow<N/2>()(x); }
};
```

- Except for odd N…

Example – computing x^N (for all N)

- For even N we need:
  return Pow<N/2>()(x)*Pow<N/2>()(x);
- For odd N we need:
  return x*Pow<(N-1)/2>()(x)*Pow<N/2>()(x);
- For N=0 we need: return 1

```
template <size_t N> struct Pow {
  double operator()(double x) const {
    return (N%2) ? x*Pow<N/2>()(x)*Pow<N/2>()(x)
                 : (N ? Pow<N/2>()(x)*Pow<N/2>()(x))
                      : 1 ); } };
```

- Both expressions in ?: must be valid, even if only one is used to compute result (cannot hide compiler errors in ?:)
  - Same for &&/|| short-circuiting

Example – computing x^N with partial specializations

```
template <size_t N, bool odd=N%2> struct Pow1 {
  double operator()(double x) const {
    return Pow1<N/2>()(x)*Pow1<N/2>()(x); }
};
template <size_t N> struct Pow1<N, true> {
  double operator()(double x) const {
     return x*Pow1<(N-1)/2>()(x)*Pow1<(N-1)/2>()(x); }
};
template <> struct Pow1<1, true> {
  double operator()(double x) const { return x; }
};
template <> struct Pow1<0, false> {
  double operator()(double x) const { return 1; }
};
```

Example – computing x^N

```
template <size_t N, bool odd=N%2> struct Pow1 { … };
template <size_t N> struct Pow {
  double operator()(double x) const {return Pow1<N>()(x);}
};
```

- Adding termination condition as a new template parameter allows us to specialize on that condition
  - Works for any numeric expression of N (could be N1, N2…)
- Using default argument saves us from [mis]typing the expression by the client (Pow() of 5, even?)
- pow(x, 20) – 9 seconds, Pow<20>()(x) – 0.05 seconds (straightforward implementation was 0.5 seconds)

More practical metaprogramming examples

- Popcount computation – number of 1's in a bit string

```
unsigned long count = 0;
for (unsigned char* c = str; *c; ++c) {
  count += poptable[*c];
};
```

- poptable[x] – number of 1's in byte x
- poptable can be easily generated by a metaprogram
- Other applications of lookup tables – CRC, encryption
- This is not complex metaprogramming, merely useful

Guidelines for using metaprogramming in everyday application programming

- Compile-time numeric calculations
  - Also consider constexpr functions in C++11
  - Use precomputed static values when it's simpler (π)
- Code generation using numeric calculations
  - Less computation at run time == good

Types of metaprograms

- Compile-time numeric calculations
- Code generation using numeric calculations
- "Self-adapting" code
  - Portability, e.g. 64-bit vs 32-bit, big-endian vs little-endian
  - Code that depends on word size or size of some type

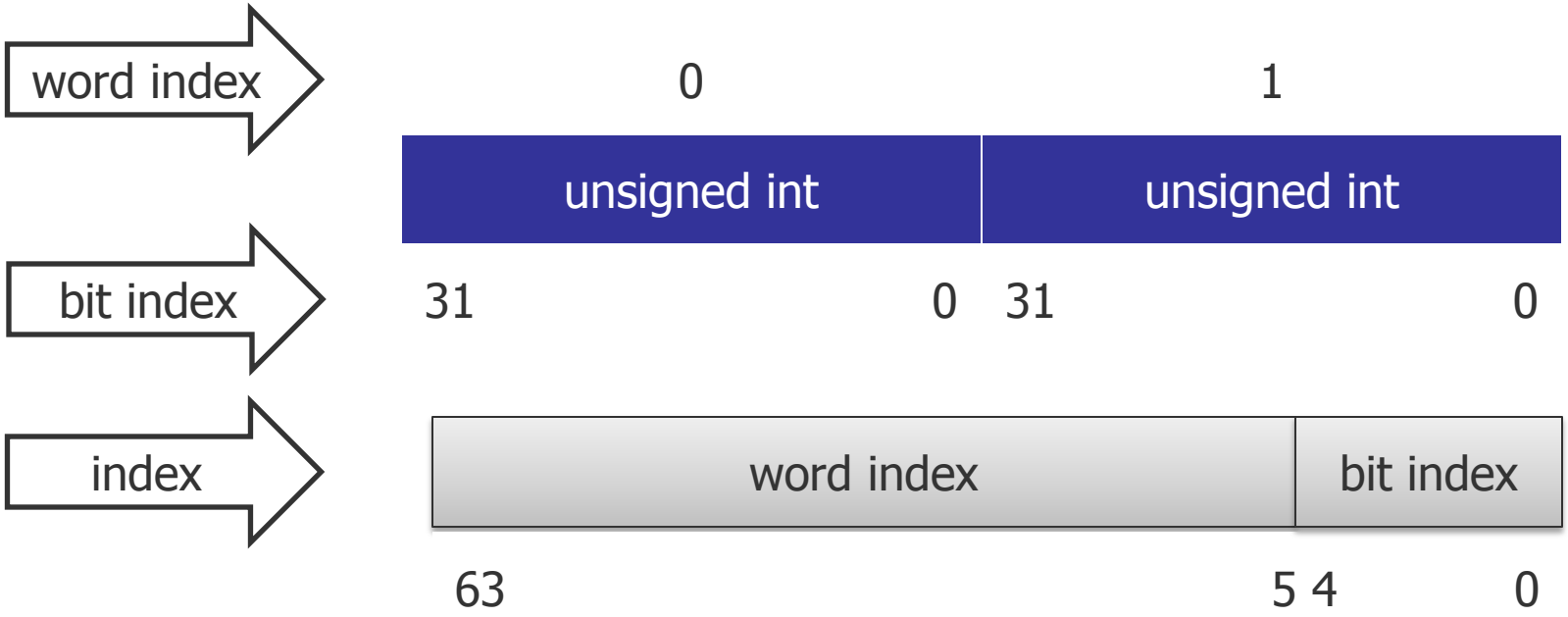
Example – bit string

- BitString is conceptually a string of bits
- BitString is implemented as an array of some basis type
  - unsigned long, unsigned int, unsigned char, m256i
- There may be reasons to use different basis types on different machines or for different applications, depending on what operations are needed and what bit manipulation instructions are available
  - Some advanced bit instructions operate only on certain types

```
template <typename T> class BitString {
  …
  T* data_;
};
```

Indexing bits

- Indexing bits is simple because sizeof(T) is a power of 2 for all integer T



bit_index = index & 0x1F; word_index = index >> 5;

Indexing bits in a template

- We want to support words of any size

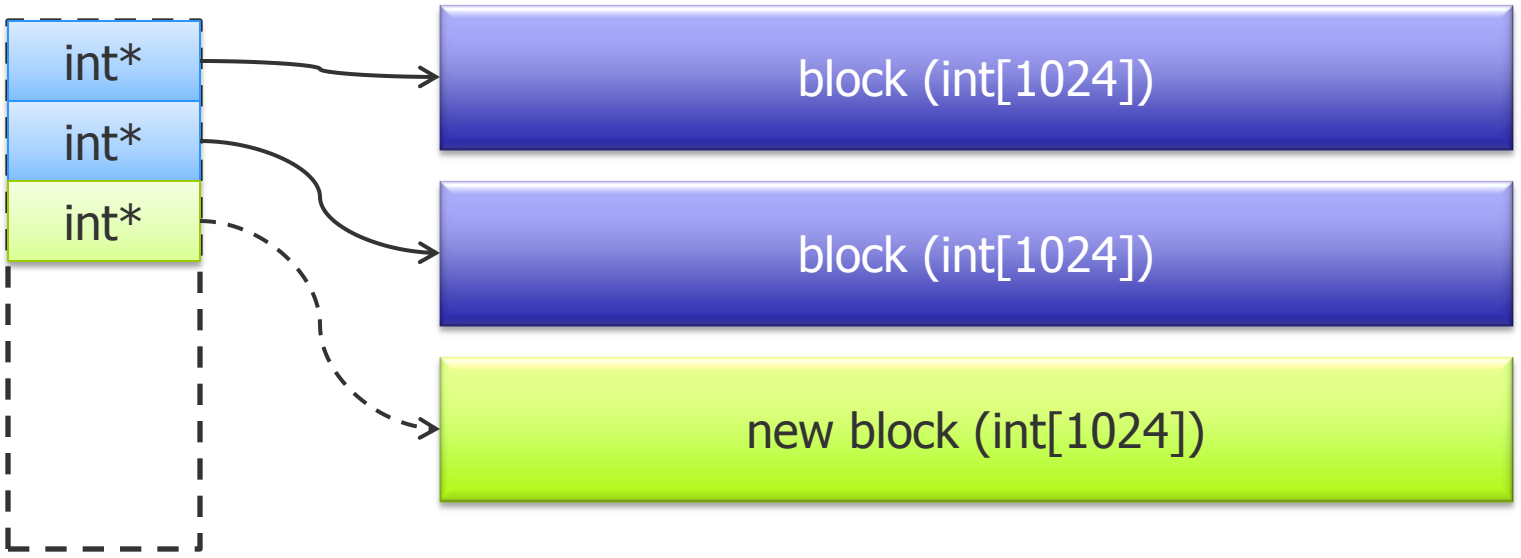
```
template <typename T> class BitString {
  T* data_;
  enum { SHIFT = Log2<sizeof(T)>::value+3 };
  enum { MASK = (T(1) << SHIFT) – 1 };
  bool get(size_t i) const {
    T word = data_[i >> SHIFT];          // Word index
    T mask = T(1) << (i & MASK);         // Bit index
    return word & mask;
  }
};
```

Example - deque

- Deque is an array-like container that can grow without reallocating the memory occupied by the data
- How did they do that?

Example – deque<int>

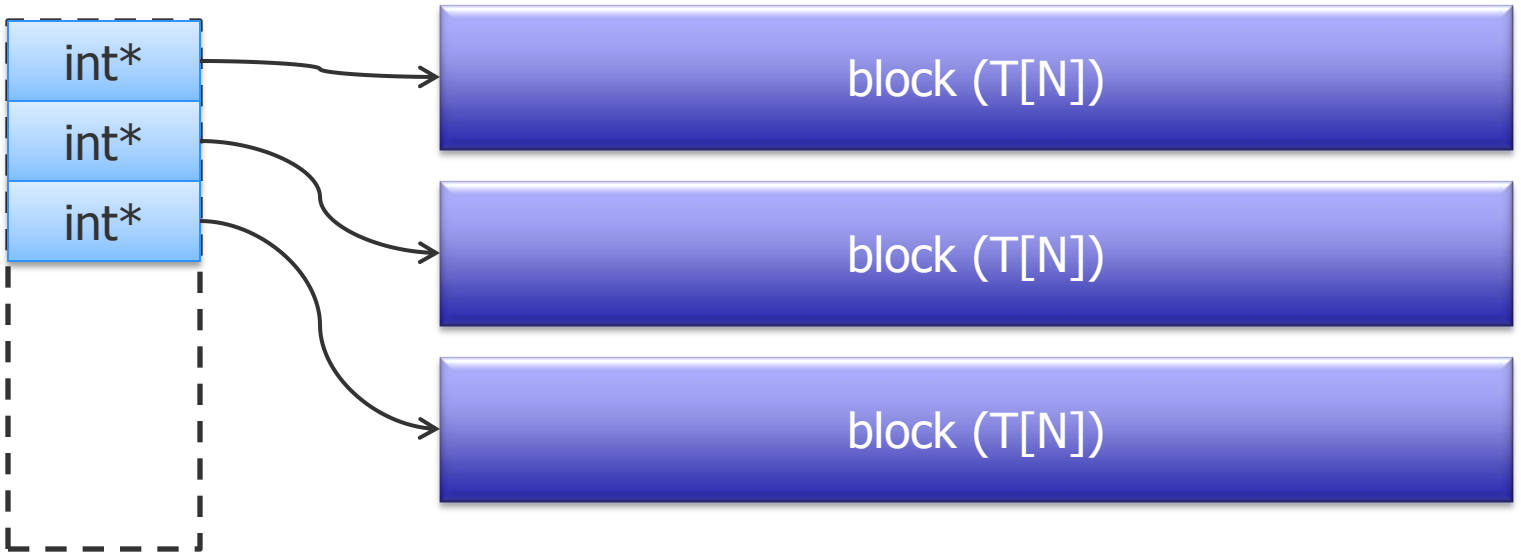
- Deque is an array-like container that can grow without reallocating the memory occupied by the data
- How did they do that?



- There are several options for managing the pointer array, that's not relevant right now

Example – deque<T>

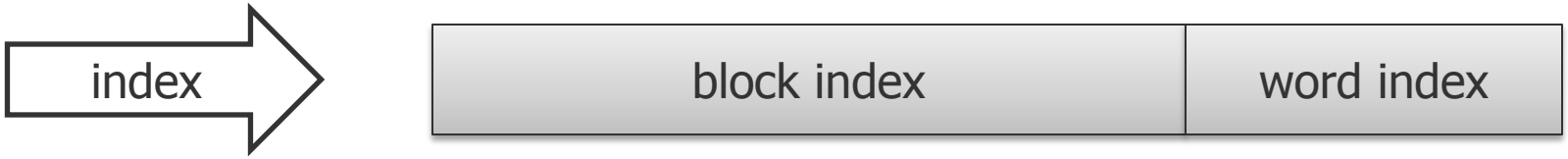
- Memory management is easier and more efficient if blocks are of the same size
- But now number of elements per block varies



- N = BlockSize/sizeof(T)

Indexing deque<T>

- If sizeof(T) is a power of 2, index can be split into block index (index in the array of pointers) and word index within the block



- BlockSize is also a power of 2, i.e. a multiple of sizeof(T)

```
size_t word_index = index & MASK;
size_t block_index = index >> SHIFT;
```

SHIFT = Log2<BlockSize/sizeof(T)>::value

MASK = (1UL << SHIFT) – 1

— Why "1UL"? "1" is int, what if SHIFT > 32?

Guidelines for using metaprogramming in everyday application programming

- Compile-time numeric calculations
  - Also consider constexpr functions in C++11
  - Use precomputed static values when it's simpler (π)
- Code generation using numeric calculations
  - Less computation at run time == good
- "Self-adapting" code
  - Automatic portability

Types of metaprograms

- Compile-time numeric calculations
- Code generation using numeric calculations
- "Self-adapting" code
- Type manipulations
  - Checking types for certain properties
  - Converting types to other types

Type manipulations

- These meta-functions operate on types
- Results are usually types
  - There are also "has-feature-x" metafunctions returning a boolean
- Is a given type a pointer?

```
template <typename T> struct IsPtr {
  enum { value = 0 };
};
template <typename T> struct IsPtr<T*> {
  enum { value = 1 };
};
```

- Is "const T*" a pointer? What about "T* const"?
- Is function pointer a pointer?

C++11 does it for you

- C++11 has is_pointer<T>
- Pointers to const/volatile are pointers
- Const/volatile pointers are pointers
- Function pointer is considered a pointer
- Member function pointer is not (there is another metafunction for that)
- If you need to write a metafunction that is not in the library, you need to think about such details

Type manipulations

- These meta-functions operate on types
- Results are usually types
  - There are also "has-feature-x" metafunctions returning a boolean
  - C++11 convention is to return both a boolean and a type
- Is a given type a pointer?

```
template <typename T> struct IsPtr {
  enum { value = 0 };
  typedef false_type type;
};
template <typename T> struct IsPtr<T*> {
  enum { value = 1 };
  typedef true_type type;
};
```

Types as boolean results

- C++11 has true_type and false_type defined and returned by metafunctions that also return booleans
- If you have to do it yourself:

```
struct TrueType {
  typedef TrueType type;
  enum { value = true };    // static constexpr in C++11
  constexpr operator bool() const { return true; } // C++11
};
struct FalseType {
  typedef FalseType type;
  enum { value = false };   // static constexpr in C++11
  constexpr operator bool() const { return false; } // C++11
};
```

Type manipulations

- These meta-functions operate on types
- Is a given type a pointer? If yes, what does it point to?

```
template <typename T> struct RemovePtr  {
  typedef T type;
};
template <typename T> struct RemovePtr<T*> {
  typedef T type;
};
```

- For a pointer type U=T*, RemovePtr<U>::type is T
- Again remember const, volatile, etc
- C++11 has remove_ptr<T>
  - For non-pointer types it returns the type itself

Life before C++11 was miserable…

- I had to write IsPtr, IsRef, RemoveRef, AddConst, HasMemberFunctionFoo, etc, all by myself
- Waist-deep in compiler bugs…

With C++11 you have it all…

- C++11 provides is_pointer, is_reference, remove_reference, add_const, has_member_fiunction_foo
  - Well, not the last one
- C++11 has a host of other useful metafunctions
  - is_same, is_assignable, is_polymorphic
  - Most can be implemented in C++03
  - Some require compiler support (has_virtual_destructor)
- Compilers are getting better too
- Still no HasMemberFunctionFoo
  - You still has to know this stuff (job security!)
- But why would you want to?

Guidelines for using metaprogramming in everyday application programming

- Compile-time numeric calculations
  - Also consider constexpr functions in C++11
  - Use precomputed static values when it's simpler (π)
- Code generation using numeric calculations
  - Less computation at run time == good
- "Self-adapting" code
  - Automatic portability
- Type manipulations
  - In C++11, use std:: metafunctions (in C++03, know C++11 additions, follow the interface, you can implement most of them)

Types of metaprograms

- Compile-time numeric calculations
- Code generation using numeric calculations
- "Self-adapting" code
- Type manipulations
- Type-dependent code
  - Code that performs similar function differently for different types

Type-dependent code

- Just like numeric calculations were useful mostly for implementing parametrized code, type operations are used for implementing type-dependent code
- The same, or logically equivalent, algorithm or container may have different implementations for different types
  - This can be done for efficiency or functionality
- Type-parametrized code is a more general case of a numerically parametrized code, distinguishing properties are more complex than type's size

Example – container with internal sort

- We have a MagicContainer<T> that maintains elements in sorted order, according to operator<()
- If the MagicContainer stores pointers, we want to compare what they point to
- Such objects usually have optional custom comparators but this requires comparators for all pointer types
  - Users forget such things

Example – container with internal sort

- Easy to implement using partial specialization and IsPtr<T> we already have (or std::is_pointer<T>)

```
// For non-pointers
template <typename T, bool Ptr> class MagicImpl {
  sort() { … T &x1, &x2; if (x1<x2) std::swap(x1, x2); … }
};
// For pointers
template <typename T> class MagicImpl<T, true> {
  sort() { … T x1, x2; if (*x1<*x2) std::swap(x1, x2); … }
};
// For all types
template <typename T> class MagicContainer<T> :
  public MagicImpl<T, IsPtr<T>::value> {};
```

Example – container with internal sort

- Another way, for all types

```
template <typename T> class MagicContainer<T> {
  typedef typename remove_pointer<T>::type U;
  // Store T but compare U
  sort() { … U &x1, &x2; … T p1, p2;
    if (x1<x2) std::swap(p1, p2); … }
};
```

- Isn't it great that remove_pointer<T> returns T for non-pointer types?
  - Not always…
- In this example, the desired functionality depends on the type: MagicContainer<T> always stores T, but compares either T or *T

Type-dependent code for efficiency

- More common are examples where different types are handled differently for efficiency
- Sort: random-access iterators support "it + n" and allow fast sort, bidirectional iterators support only "it++" and necessitate slow sort
  - Usually done with traits but requires iterators to be compliant
  - In practice, may require even larger changes to the algorithm
- General algorithms that call member functions for some tasks when such are available
- The principle is similar, use partial specialization and "HasFeature" metafunctions
  - Usually not all code is duplicated, much is reusable
- How do you test for existence of a member function?

Example – use T::sort() when available

- SortingProcessor<T> calls T::sort() if one exists, otherwise sorts the sequence [T::begin(), T::end())
- Easy to do using partial specialization and HasMemberFunctionSort (if we have one…)

```
template <class T, bool HasSort> class ProcessorImpl {
  Process() { … T& obj; std::sort(obj.begin(), obj.end()); … }
};
template <class T> class ProcessorImpl <T, true> {
  Process() { … T& obj; obj.sort(); … }
};
template <class T> class SortingProcessor :
  ProcessorImpl<T, HasMemberFunctionSort<T>::value> {}
```

How to test for something that may not be there?

- The problem: to check for existence of a member function, we need to somehow refer to its name
- If the type does not have such member function, any code that refers to that name will not compile
- We want the overall code to compile, just differently
- We need some context in which compilation failures are not fatal
- Standard says nothing about what "compiles" – it defines valid and invalid expressions. We need a context in which an invalid expression does not require diagnostic
  - This note was brought to you by the union of language lawyers

SFINAE – where compiler errors go to hide

- **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror
- Context: function overload resolution
  - All functions with the given name and the right number of arguments are considered as potential overloads (*)
  - This includes template and non-template functions
  - Template functions must have types of their arguments deduced: template parameters are substituted by specific arguments
  - If this substitution results in an invalid expression, type deduction fails and the function is not considered an overload candidate
    - If the invalid expression is the argument type or return type, diagnostic is not issued (*) and overload resolution proceeds
  - Of all overload candidates, the best match is chosen (*)

* Usually this works out the way you'd expect

  - If it does not, figuring out what really happens would take more time than this entire class

SFINAE check for a member function

- We need two overloaded template functions
  - First should contain some expression referring to the member function and be a better match for the call
  - Second should always match the call, not as well as the first
- The first function will be selected by overload resolution as long as the expression is valid
- We need to test which function was selected
- We need to avoid actually calling the functions
- Really, we need to avoid evaluating expressions even at compile-time: if the expression must be evaluated but is invalid, diagnostic is issued

Argument non-evaluation

- Operands of sizeof, decltype, declval, alignof, typeid, and noexcept are never evaluated!
- Not even at compile time
  - This deserves an example:
  - You can't use sizeof() on incomplete types

```
class Foo;
template <typename T> struct MakePtr {
  typedef T* type;
};
sizeof(MakePtr<Foo>::type) // Valid constexpr expression

template <size_t N> class Bar { int a[N]; };      // N must be
Bar<sizeof(MakePtr<foo>::type)> bar;              // constexpr
```

SFINAE check for a member function

```
template <typename T> struct HasSort {
  typedef char yes;
  typedef char no[2];
  template <typename U> static auto test( void* ) ->
      decltype ( &T::sort , yes() );
  template <typename U> static no& test( … );
   enum { value = (sizeof( HasSort::test<T>(NULL) ==
                        sizeof( HasSort::yes )); };
};
class MyStorage { … void sort(); };
HasSort<MyStorage>::value        // Compiles, value == 1
HasSort<int>::value              // Compiles! value == 0
```

SFINAE before C++11

- In C++03 we have to abuse sizeof()

```
template <typename T> struct HasSort {
  typedef char yes;
  typedef char no[2];
  template <typename U>
      static yes& test( char (*)[ sizeof ( &T::sort ) ]);
  template <typename U> static no& test( … );
  enum { value = (sizeof( HasSort::test<T>(NULL) ==
                          sizeof( HasSort::yes )); };
};
class MyStorage { … void sort(); };
HasSort<MyStorage>::value        // Compiles, value == 1
HasSort<int>::value              // Compiles! value == 0
```

Example – use T::sort() when available

```
template <class T, bool HasSort> class ProcessorImpl {
  Process() { … T& obj; std::sort(obj.begin(), obj.end()); … }
};
template <class T> class ProcessorImpl <T, true> {
  Process() { … T& obj; obj.sort(); … }
};
template <class T> class SortingProcessor :
  ProcessorImpl<T, HasMemberFunctionSort<T>::value> {}

class NoSort { … };                          // std::sort (!)
class WithSort { … void sort(); … }          // WithSort::Sort (!)

class OtherSort { … void sort(int i); … }  // ?
```

- HasMemberFunctionSort<T> checks for any member function named "sort" (ProcessorImpl expects sort(void))

Back to the drawing board

- One can use similar technique to test for a member function with a specific signature, e.g.

HasSort<T>::value == 1 iff T has "void T::sort(int k)"

  - k may be the index of the field in the record to use for sort

- What about "bool T::sort(int)"? This sort likely would work, but HasSort<> now fails
- What if T has "void T::sort(long k)"? We should use it, but HasSort<> again fails
- Checking for a method name is too broad
- Checking for a method signature is too specific

Even farther back

- Ok, our SortingProcessor is in trouble, but the MagicContainer is still good, right?
  - Stores T, compares *T for pointers, otherwise T
- What about MagicContainer<shared_ptr<U>>?
- shared_ptr<U> is an object, not a pointer – we get generic version of MagicContainer
- But shared_ptr<U> should behave like a pointer!
- What about MagicContainer<my_iterator<U>>?
- Ok, so we also need to check for U::operator*()
  - Good thing we know how to check for member functions…
- Is that the only way to create a pointer-like object?
- Non-member operator*()?

Nobody expects global operators!

- How would we check if "T + int" is a valid operation?
  - ::operator+(const T&, int)
  - ::operator+(const T&, const T&) and T(int) (not explict!)
  - T::operator+(int)
  - ::operator+(const T&, long)
  - template <typename U> ::operator+(const T&, const U&)
  - ::operator+(const T&, const T&) and template<class U> T(U)
- That's a lot of things to check for (is there more?)
  - T::operator int()?
- What we really want is to check whether "x+1" is a valid expression for "x" of type "T"
  - "Does x+1 compile?"
  - For any reason whatsoever...
- Is "*p" valid for "T p;"? Is "c.sort(1)" valid for "T& c;"?

Step back from the abyss

- We can use SFINAE to test for “valid expression”!
- We can package the solution to be easy to use
- It can be done in C++03 (C++11 removes some hacks)

Valid expression test in C++03 ("if_compiles")

```
template <typename U1> struct IsPtr {
  template <typename U> static U& DeclVal();
  typedef char yes;
  typedef char no[2];
  template <typename T1> static yes& f1( T1& x1,
      char (*a)[ sizeof(*x1) ] = NULL);
  template <typename T1> static no& f1(...);
  enum { value =
    sizeof( IsPtr::f1<U1>( IsPtr::DeclVal<U1>() ) ) ==
    sizeof( IsPtr::yes ) };
};
IsPtr<int*>::value == 1
IsPtr<int>::value == 0
IsPtr<SmartPtr<int> >::value == 1
```

“if_compiles” in C++03, wrapped up

```
#define DEFINE_IF_COMPILES( NAME , EXPR ) \
template <typename U1> struct NAME { \
  template <typename U> static U& DeclVal(); \
  typedef char yes; \
  typedef char no[2]; \
  template <typename T1> static yes& f1(T1& x1, \
      char (*a)[sizeof( EXPR )] = NULL); \
  template <typename T1> static no& f1(...); \
  enum { value = \
    sizeof(NAME::f1<U1>(NAME::DeclVal<U1>(), U1))) == \
    sizeof(NAME::yes) }; \
}
DEFINE_IF_COMPILES1(IsPtr, *x1); // Convention: x1, T1
IsPtr<int*>::value == 1
```

Valid expression test in C++03 ("if_compiles")

```
template <typename U1> struct IsPtr {
  template <typename U> static U& DeclVal();
  typedef char yes;
  typedef char no[2];
  template <typename T1> static yes& f1( T1& x1,
      char (*a)[ sizeof(*x1) ] = NULL);
  template <typename T1> static no& f1(...);
  enum { value =
    sizeof( IsPtr::f1<U1>( IsPtr::DeclVal<U1>() ) ) ==
    sizeof( IsPtr::yes ) };
};
IsPtr<int*>::value == 1
IsPtr<int>::value == 0
IsPtr<SmartPtr<int> >::value == 1
```

What can "if_compiles" do?

```
DEFINE_IF_COMPILES1(AddInt, x1+1);
DEFINE_IF_COMPILES1(DefaultCtor, new T1);
DEFINE_IF_COMPILES1(AssignInt, x1=1);
DEFINE_IF_COMPILES1(HasSort, x1.sort());
DEFINE_IF_COMPILES1(HasSortInt, x1.sort(1));
DEFINE_IF_COMPILES1(HasAnySort, &T1::sort);
```

- Checks for expressions with 2, 3, … different types are similarly defined (naming: T1, x1, T2, x2, …)

```
DEFINE_IF_COMPILES2(Add, x1+x2);
DEFINE_IF_COMPILES2(ConvertFrom, T1(x2));
DEFINE_IF_COMPILES2(Assign, x1=x2);
DEFINE_IF_COMPILES3(MyExpr, x1+x2*x3/2);
   STATIC_ASSERT(MyExpr<int, MyFrac, double>::value);
```

Valid expression test in C++11

```
#define DEFINE_IF_COMPILES( NAME , EXPR ) \
template <typename U1> struct NAME { \
  typedef char yes; typedef char no[2]; \
  template <typename T1> static auto f1(T1& x1) -> \
     decltype( EXPR ) , yes() ); \
  template <typename T1> static no& f1(...); \
  enum { value = \
    sizeof(NAME::f1<U1>(std::declval<U1&>(), U1))) == \
    sizeof(NAME::yes) }; \
}
DEFINE_IF_COMPILES1(IsPtr, *x1); // Convention: x1, T1
IsPtr<int*>::value == 1
static_assert(MyExpr<int, MyFrac, double>::value,
                  "MyExpr needs more operators");
```

Types of metaprograms

- Compile-time numeric calculations
- Code generation using numeric calculations
- "Self-adapting" code
- Type manipulations
- Type-dependent code

- Precise type processing
- Typelists and other "type collections"
- Code generation driven by type collections, e.g. "virtual template functions"
- Domain languages and tools

Practical/Applications Advanced/Libraries

Guidelines for using metaprogramming in everyday application programming

- Compile-time numeric calculations
  - Also consider constexpr functions in C++11
  - Use precomputed static values when it's simpler (π)
- Code generation using numeric calculations
  - Less computation at run time == good
- "Self-adapting" code
  - Automatic portability
- Type manipulations
  - In C++11, use std:: metafunctions (in C++03, know C++11 additions, follow the interface, you can implement most of them)
- Type-dependent code
  - Use type analysis metafunctions and partial specialization when solution depends on a specific type property
  - Use SFINAE ("if_compiles") when solution depends on supported operations

Mentor Graphics®

www.mentor.com