New Tools for Class and Library Authors

By Scott Schurr for C++ Now! 2012

Topics

- _Pragma
- static_assert
- explicit conversion operators
- decltype
- constexpr
- variadic templates

_Pragma

- Borrowed from C99
- Allows #pragma use inside a macro
- Handy for multi-compiler support

_Pragma

```
#ifdef __ADSPTS201__            // This works a treat
#define TIGER_SHARC_PRAGMA( ARG ) _Pragma ( ARG )
#else
#define TIGER_SHARC_PRAGMA( ARG )
#endif // __ADSPTS201__

#ifdef _WIN32                   // This will eventually...
#define WIN32_PRAGMA ( ARG ) _Pragma ( ARG)
#else
#define WIN32_PRAGMA( ARG )
#endif // _WIN32
```

_Pragma

You choose...

```
TIGER_SHARC_PRAGMA("section(\"IRDelay\")")
float IRDelay[1024]
```

...or...

```
#ifdef  __ADSPTS201__
#pragma section ("IRDelay")
#endif
float IRDelay[1024]
```

_Pragma

```
WIN32_PRAGMA("warning(push)")   // eventually...
WIN32_PRAGMA("warning( disable : 4068 )")
...
WIN32_PRAGMA("warning(pop)")
```

...or...

```
#ifdef  _WIN32
#pragma warning(push)
#pragma warning( disable : 4068 )
#endif
...
#ifdef _WIN32
#pragma warning(pop)
#endif
```

static_assert

C++11's BOOST_STATIC_ASSERT

"Prefer compile-time and link-time errors to runtime errors." Meyers *Effective C*++ Item 46

"Prefer compile- and link-time errors to run-time errors."
Sutter and Alexandrescu *C*++ *Coding Standards* Item 14

static_assert

Takes two parameters:

- A test that can be resolved at compile time, and
- Any kind of string literal used as the diagnostic message

```
static_assert(sizeof(void*) == sizeof(long),
  "Pointers and longs are different sizes");
```

static_assert

The following macro makes static_assert easy to use for the bulk of cases

```
#define STATIC_ASSERT(...) \
  static_assert(__VA_ARGS__, #__VA_ARGS__)
```

```
STATIC_ASSERT(sizeof(void*) == sizeof(char));
```

```
static assertion failed: sizeof(void*) == sizeof(char)
```

Note that C++11 supports variadic macros!

Explicit conversion operators

"Be wary of user-defined conversion functions"

Meyers *More Effective C++* Item 5

"Avoid providing implicit conversions"

Sutter and Alexandrescu *C++ Coding Standards* Item 40

Explicit conversion operators

Just like constructors, conversion operators can now be explicit

```
class MyClass {
public:
   explicit MyClass(int i);
   explicit operator std::string() const;
};
```

Explicit conversion operators

You must cast to use the explicit conversion

```
void use_mc(const MyClass& m);
int i;
...
use_mc(i);                                  // fails
use_mc(static_cast<MyClass>(i));            // works

void use_str(const std::string& s);
MyClass mc(1);
...
use_str(mc);                                // fails
use_str(static_cast<std::string>(mc));      // works
```

Explicit bool conversion

Explicit bool conversion is special…

```
class MyClass {
public:
  explicit MyClass(int i);
  explicit operator bool() const;
};

const MyClass mc(1);
if (mc)  { }                        // works
const int  j = mc ? 5 : 10;         // works
const bool b = mc;                  // fails
```

Explicit conversion summary

- I like explicit bool conversion operators. I'll use them.
- For non-bool I'll still prefer named functions to conversion operators, but…
- When generics need type conversions I'll use explicit conversion operators.

decltype

Yields the type of an expression without evaluating it. Similar to sizeof.

```
#include <type_traits>
using namespace std;

template <typename Ta, typename Tb>
auto mult(const Ta& a, const Tb& b) -> decltype(a * b);

STATIC_ASSERT(is_same<decltype(mult('a',3)),int>::value);

STATIC_ASSERT(is_same<decltype(mult(2,3.7)),double>::value);
```

decltype test bed

The following macro helps while messing with types

```
#include <type_traits>
#define IS_SAME_TYPE(T1, T2) \
static_assert(std::is_same< T1, T2 >::value, \
"\n'Cuz " #T1 " and " #T2 " are not the same type.")
```

decltype(e) rule 1

If e is a function call or an overloaded operator invocation, the result is the declared return type

```
int int_ret();
IS_SAME_TYPE(decltype(int_ret()),     int);

const int&& rref_ret(int i);
IS_SAME_TYPE(decltype(rref_ret(1)),   const int&&);

const char&& rref_ret(char c);
IS_SAME_TYPE(decltype(rref_ret('a')), const char&&);
```

decltype(e) rule 2

If e refers to a variable in

- local scope,
- namespace scope,
- a static member variable, or
- a function parameter then

The result is the parameter or variable's declared type

decltype(e) rule 2 examples

```
int i;
IS_SAME_TYPE(decltype(i),      int);

struct s_t { char c; double d; };
s_t S;
IS_SAME_TYPE(decltype(S),      s_t);
IS_SAME_TYPE(decltype(S.c),    char);

s_t* S_p = &S;
IS_SAME_TYPE(decltype(S_p),    s_t*);
IS_SAME_TYPE(decltype(S_p->d), double);
```

decltype(e) rules 3 and 4

3. If rules 1 and 2 don’t apply and e is an lvalue, if e is of type T, the result is T&.

```
int* p;
IS_SAME_TYPE(decltype(*p), int&); // * returns an lvalue
```

4. If rules 1 and 2 don’t apply and e is an rvalue, if e is of type T, the result is T.

```
IS_SAME_TYPE(decltype(1+2), int); // + returns an rvalue
```

decltype rule 5

Extra parentheses around expression e [e.g., decltype((e))] produce a "reference to the type of" e.

```
int i;
IS_SAME_TYPE(decltype((i)),        int&);

struct s_t { char c; double d };
s_t* S_p = &S;
IS_SAME_TYPE(decltype((S_p->d)),  double&);

const s_t* Sc_p = &S;
IS_SAME_TYPE(decltype((Sc_p->d)), const double&); // !
```

decltype helpers

When using decltype, it's good to know about these from <type_traits>…

```
std::remove_reference
std::add_lvalue_reference
std::add_rvalue_reference
std::remove_cv
std::remove_const
std::remove_volatile
std::add_cv
std::add_const
std::add_volatile
```

decltype summary

- decltype produces (almost always) the "declared type".
- Remember that and look up the other rules if you get stuck.
- Use the helpers if decltype produces almost what you want.

constexpr

Never put off till tomorrow what you can do today

Thomas Jefferson

Prefer compile time computations over run time computations

Someone Famous should say this in *Some Book*

constexpr

- Generalized constant expressions
  - static const int on steroids

- Use in two contexts:
  - Variable declarations, and
  - Function or constructor declarations

constexpr example

```
template <typename T = double>
constexpr T eulers_num()
{ return static_cast<T>(2.718281828459045235360287471); }

constexpr double e_d = eulers_num();
constexpr float  e_f = eulers_num<float>();
constexpr int    e_i = eulers_num<int>();

// The following happens at compile time!
static_assert (e_d != e_f, "Precision matters!");
static_assert (e_i <  e_f, "Precision really matters!");
```

constexpr variables rules

- Must be immediately constructed or assigned a value
- May only be assigned…
  - literal values,
  - constexpr values, or the
  - return value of constexpr functions
- For user defined types, must invoke a constexpr constructor

constexpr function rules

- May not be virtual
- Must return a literal or constexpr type
- Each of its parameters must be literal or constexpr types
- The body may contain:
  - Safe stuff (static_assert, typedefs, using…)
  - Exactly one return statement that contains only literals, constexpr variables and functions.
  - Note: un-evaluated AND, OR, and ?: subexpressions are not considered.

Let's make something

```
class str_const {                       // constexpr string
private:
   const char* const p_;
   const std::size_t  sz_;
public:
   template<std::size_t N>
   constexpr str_const(const char(&a)[N]) :      // ctor
       p_(a), sz_(N-1) {}
   constexpr char operator[](std::size_t n) {    // []
       return n < sz_ ? p_[n] :
             throw std::out_of_range("");
   }
   constexpr std::size_t size() { return sz_; } // size()
};
```

Hold yer horses!

A throw in a constexpr function?

- Yep. The throw must be unevaluated under compile-able conditions. But if you want to prevent something from compiling, you throw.
- Constexpr functions called with non-literals turn into plain-old functions. At runtime a throw is the right approach.

What can a str_const do?

```
constexpr str_const test("Hi Mom!");

STATIC_ASSERT(test.size() == 7);
STATIC_ASSERT(test[6] == '!');
```

We can now examine the contents and length of a c-string at compile time.

So what?

Let's make something else

```
template <typename T = unsigned int>
constexpr T binary_const(
   str_const b,
   std::size_t n = 0,
   T x = 0)
{
   return
       n == b.size() ? x :
       b[n] == ',' ? binary_const<T>(b, n+1,  x) :
       b[n] == ' ' ? binary_const<T>(b, n+1,  x) :
       b[n] == '0' ? binary_const<T>(b, n+1, (x*2)+0) :
       b[n] == '1' ? binary_const<T>(b, n+1, (x*2)+1) :
       throw std::domain_error(
              "Only '0', '1', ',', and ' ' may be used");
}
```

We have a binary literal!

```
// Binary conversion at compile time
using u32_t = unsigned int;
constexpr u32_t i_maskA =
  binary_const("1111,1111,1110,0000,0000,0000,0000,0000");
constexpr u32_t i_maskB =
  binary_const("0000,0000,0001,1111,1111,1000,0000,0000");
constexpr u32_t i_maskC =
  binary_const("0000,0000,0000,0000,0000,0111,1111,1111");

STATIC_ASSERT(i_maskB == 0x001FF800);
STATIC_ASSERT(i_maskA + i_maskB + i_maskC == 0xFFFFFFFF);
```

…and the generated code?

g++ 4.7.0 for Cygnus on Windows XP 32-bit produces only three instructions:

```
movl $-2097152, 12(%esp)
movl $2095104, 8(%esp)
movl $2047, 4(%esp)
```

Compile-time word-size validation should be added. [homework?]

More binary literals

```
using u8_t = unsigned char;
constexpr u8_t b_maskA = binary_const<u8_t>("1110 0000");
constexpr u8_t b_maskB = binary_const<u8_t>("0001 1000");
constexpr u8_t b_maskC = binary_const<u8_t>("0000 0110");
constexpr u8_t b_maskD = binary_const<u8_t>("0000 0001");

STATIC_ASSERT(
  b_maskA + b_maskB + b_maskC + b_maskD == 0xFF);

constexpr double d = binary_const<double>("1000");
STATIC_ASSERT(d == 8);
```

constexpr considerations

- Can’t examine the internals of a floating point value

- Floating point calculation results may differ between compile-time and runtime

- The compiler may evaluate at runtime! (Ouch!)

constexpr summary

- I'll use 'em everywhere I can.
- Declare variables constexpr to avoid runtime costs.
- Many compile-time algorithms make poor runtime algorithms. Use caution.
- Evaluation at runtime is a quality of implementation issue. Know your compilers!

Variadic templates

- Fix an irritant where C++98 templates required explicit specialization for various numbers of template arguments

- Open the door for new idioms

2 kinds of variadic templates

- Class templates have only types

```
template <class... Types> class tuple;
template <class T, class... Args> struct is_constructible;
```

- Function templates have types and parameters

```
template <class... Types>
tuple<VTypes...> make_tuple(Types&&... values);

template <class L1, class L2, class... L3>
int try_lock(L1& lock1, L2& lock2, L3&... moreLocks);
```

- The “...” declares a parameter pack

2 kinds of parameter packs

- Variadic class templates need:
  - a template parameter pack

```
template < class... Types > class tuple;
```

- Variadic function templates need:
  - a template parameter pack for types and
  - a function parameter pack for values

```
template< class T, class... ParamTypes >
shared_ptr<T> make_shared( ParamTypes&&... params );
```

But what *is* a parameter pack?

- It's just a notation to the compiler
- The compiler replaces parameter packs with:
  - 0 to n types (template param packs)
  - 0 to n arguments (function param packs)
- All parameter packs are explicit complete types before the linker sees them.

Class template parameter packs

- May have at most one parameter pack
- The parameter pack must be the last template argument

```
template <class T, class... Args> struct is_constructible;
```

Function template param packs

The function parameter pack is

- a function parameter declaration containing a template parameter pack expansion.
- It must be the last parameter in the function parameter list.

The template parameter pack

- Contains the types of the parameters
- May have multiple parameter packs, e.g.

```
template<class... Ts, class... Us>
bool operator==(const tuple<Ts...>& t, const tuple<Us...>& u);
```

The two packs are always in lock step

The simplest example

```
template <typename T1, typename T2, typename T3>
void OldStyleVariadic(T1 p1, T2 p2, T3 p3)
{ std::cout << "2 args" << std::endl << std::ends; }
// ...
void OldStyleVariadic()
{ std::cout << "0 args" << std::endl << std::ends; }

template <typename... Ts> void NewStyleVariadic(Ts... vs)
{  OldStyleVariadic(vs...); }

int main() {
   NewStyleVariadic(1.0, 2, "3");
   NewStyleVariadic();
   return 0;
}
```

Prints

```
$ variadic_pass.exe
2 args
0 args
```

A better example

```
// Print no object.  Terminates recursion.
void variadic_cout()
{ std::cout << std::endl << std::ends; }

// Print the first object, pass the rest.
template <
  typename T,                     // type of 1st object
  typename... TRest>              // types of the rest
void variadic_cout(
  const T& obj,                   // 1st object
  const TRest&... rest)           // the rest of them
{
  std::cout << obj << " ";        // print 1st object
  variadic_cout(rest...);         // print the rest
}
```

variadic_cout in action

Program

```
int main()
{
  const std::string attrib("Gee, thanks Mr. Meyers!");
  variadic_cout("Look!", 3.4, '&', 48, "work!", attrib);
  return 0;
}
```

Prints

```
$ variadic_cout.exe
Look! 3.4 & 48 work! Gee, thanks Mr. Meyers!
```

Recursion of variadic_cout

```
variadic_cout(obj = "Look!", rest = {3.4, ... , attrib})
{ cout << obj;                       // "Look!"
  variadic_cout(obj = 3.4, rest = {'&', ... , attrib})
  { cout << obj;                     // "3.4"
    variadic_cout(obj = '&', rest = {48, ..., attrib})
    { cout << obj;                   // "&"
      variadic_cout(obj = 48, rest = {"work!", attrib})
      { cout << obj;                 // "48"
        variadic_cout(obj = "work!", rest = {attrib})
        { cout << obj;               // "work!"
          variadic_cout(obj = attrib, rest = {})
          { cout << obj;             // "Gee, thanks..."
            variadic_cout()          // Ends recursion
            { cout << endl << ends; }}}}}}}
```

Building a parameter pack

Function parameter packs build themselves

```
variadic_cout(3*3, '+', 4.0f*4, '=', 5*5.0);
```

If you already have a function parameter pack you can add to either or both ends. But no insertions.

```
template <typename... Ts>
void add_quotes(Ts... args)
{ variadic_cout('"', args..., '"'); }
```

Pulling from a parameter pack

- The compiler assigns types and parameters from left to right

- So you can only remove parameters from the left. But you can remove as many at one time as you want.

Variadic min

```
// Recursion termination
template <typename T>
const T& mono_type_min(const T& a)
{
    return a;               // recursion termination
}

// Recursive part
template <typename T, typename... TRest>
const T& mono_type_min(const T& a, const TRest&... rest)
{
    return std::min(a, mono_type_min(rest...));
}
// std::min<> takes one type. All types must be the same.
```

Improved variadic min

Improvements?

- Support multiple types
- Require a minimum of 2 arguments

The code gets bigger, so we'll take two slides…

Variadic min recursion

```
#include <type_traits>              // std::common_type

template <typename L, typename R, typename... Ts,
   typename C = const typename    // C is best return type
      std::common_type<L, R, Ts...>::type>
C common_type_min(
   const L& lhs,                  // Explicit lhs and rhs
   const R& rhs,                  //   requires 2 args
   const Ts&... rest)
{
    return std::min<C>(lhs, common_type_min(rhs, rest...));
}
```

Variadic min termination

```
template <typename L, typename R,
   typename C =
       const typename std::common_type<L, R>::type>
C common_type_min(const L& lhs, const R& rhs)
{
    return std::min<C>(lhs, rhs);
}

// Terminates recursion because a non-variadic
// template is a better match than a variadic template
```

Let's try it

```
int main()
{
   const short s = 21;
   auto minTest = common_type_min(20, 16.0, 14.0f, 'c', s);
   assert(minTest == 14.0);
   IS_SAME_TYPE(decltype(minTest), double);

   return 0;
}
```

Looks like it works

Is this production code?

Neither of these are production examples

- Use std::min(), not mono_type_min()

```
assert(6 == std::min( {7, 6, 10} ));
assert('!' == std::min( {'a', '!', ';', '0', '5'} ));
```

- common_type_min() has too many temporaries and copies to be efficient

But they may be starting points…

Variadic template classes

The end goal is a variadic_log class

- Captures values of user-specified types quickly
- Prints them later
- I use a pre-C++11 version for errors

We'll approach it in parts:

1. variadic_capture
2. printf_tuple
3. variadic_log

variadic_capture pt. 1

```
// Capture a bunch of values to save for later.
template <typename... Ts>
class variadic_capture
{
private:
  using storage_t = std::tuple<Ts...>;
  storage_t storage_;

public:
  // Capture values.  Doesn’t look type safe, but it is.
  void capture(Ts... vs)
  { storage_ = std::make_tuple(vs...); }
```

variadic_capture pt. 2

```
  // Number of elements we always capture.
  constexpr std::size_t count() const
  { return sizeof...(Ts); }

  // Get an entry.  Index must be known at compile time!
  template <std::size_t index>
  auto get() ->
      const typename
             std::tuple_element<index, storage_t>::type&
  { return (std::get<index>(storage_)); }
};
```

Using variadic_capture

```
// Make a variadic_capture
variadic_capture<int, const char*, double> capt;

// Check size at compile time!
STATIC_ASSERT(3 == capt.count());

// Populate a variadic_capture
constexpr char say[] = "say";
capt.capture(3, say, 5.9);        // Compile-time typesafe

// Retrieve contents.
assert(3       == capt.get<0>());
assert(&say[0] == capt.get<1>());
assert(5.9     == capt.get<2>());
```

It’s a wrapper around tuple

- Not very friendly.
- Particularly, get<n> index must be known at compile time.

Maybe things will get better.

Can we printf a tuple’s contents? 3 parts.

tuple_unroller pt. 1

```
template <std::size_t N> class tuple_unroller
{
public:
  template<typename... TTup, typename... TArgs>
  static void printf(
      const char* format,
      const std::tuple<TTup...>& storage,
      const TArgs... args)
  {
      STATIC_ASSERT(N <= sizeof...(TTup));
      const auto arg = std::get<N-1>(storage);
      tuple_unroller<N-1>::       // class recursion
            printf(format, storage, arg, args...);
  }
};
```

tuple_unroller pt. 2

```
// Specialization to stop recursion and print
template <> class tuple_unroller<0>
{
public:
   template<typename... TTup, typename... TArgs>
   static void printf(
        const char* format,
        const std::tuple<TTup...>& storage,
        const TArgs... args)
   {
        std::printf(format, args...);
   }
};
```

printf_tuple

```
// Friendly interface to do the printf
template <typename... TTup>
void printf_tuple(
  const char* format,
  const std::tuple<TTup...>& storage)
{
  tuple_unroller<sizeof...(TTup)>::
      printf(format, storage);
}
```

Using printf_tuple

Program

```
int main()
{
  auto test_tup = std::make_tuple("chair", 4, "legs");
  printf_tuple("My %s has %d %s\n", test_tup)
  return 0;
}
```

Prints

```
$ printf_tuple.exe
My chair has 4 legs
```

Looks promising

printf_tuple was…

- Easy to use
- Unsurprising

Now for the final goal

variadic_log in 3 parts

variadic_log pt. 1

```
template <typename... Ts> class variadic_log
{
private:
   volatile bool fired_;                        // have we fired?
   const std::string format_;                   // printf format
   std::tuple<Ts...> storage_;                  // value storage

public:
   // Constructor collects the printf format string
   variadic_log(const std::string& format_str) :
       fired_(false),
       format_(format_str),
       storage_()
   { }
```

variadic_log pt. 2

```
// fire() collects values and notes that we fired
void fire(Ts... vs)             // args are typesafe
{
    if (!fired_) {
        storage_ = std::make_tuple(vs...);
        fired_ = true;
    }
}
```

variadic_log pt. 3

```
    // If fire()d then print.  Else just return.
    void print_if_fired()
    {
        if (fired_) {
            printf_tuple(format_.c_str(), storage_);
            fired_ = false;
        }
    }
};
```

Using variadic_log

```
int main()
{
   // Make a variadic_log
   variadic_log<const char*, double, double>
       log("Cabin %s of %.1f exceeds %.1f");

   // Fire the variadic_log (from another thread)
   constexpr char temp[] = "temperature";
   log.fire(temp, 180, 72);

   // Print the log
   log.print_if_fired();    return 0;
}
```

```
$ variadic_log.exe
Cabin temperature of 180.0 exceeds 72.0
```

Variadic templates summary

- Learn to love recursion
- Sweet, once you get the hang of it
- Watch for code bloat!
- I want the next version of *C++ Templates* by Vandevoorde and Josuttis

What we covered

- _Pragma
- static_assert
- explicit conversion operators
- decltype
- constexpr
- variadic templates

Sources

- Scott Meyers, *Overview of the New C++* January 6th 2012
- Pete Becker et. al., *Working Draft, Standard for Programming Language C++* February 28th 2011
- str_const adapted from http://en.cppreference.com/w/cpp/language/constexpr
- Andre Alexandrescu *C++ and Beyond 2011* Slides, *Variadic Templates*
- Variadic min adapted from: Nordlow and Andre Caron http://stackoverflow.com/questions/7539857/reflections-on-c-variadic-templated-versions-of-stdmin-and-stdmax
- Tuple unrolling adapted from: David at http://stackoverflow.com/questions/687490/how-do-i-expand-a-tuple-into-variadic-template-functions-arguments

Questions?

Thanks for attending