Selected Topics in Computer Programming #3

Edition: 2009-07-22

Objectives:

- To introduce and apply the syntax and semantics of C++ function templates and class templates.
- To understand common template-based programming practices and techniques, including generic programming, traits, concepts, and adapters.
- To understand and apply techniques of C++ template metaprogramming, including compile-time type computations, algorithms, and control structures.

Selected Topics in Computer Programming #3

C++ Templates and Template Metaprogramming

Walter E. Brown, Ph.D. <wb@fnal.gov>
Computing Division
Fermi National Accelerator Laboratory

A little about me

- B.A. (mathematics); M.S., Ph.D. (computer science).
- Professional programmer for nearly 40 years.
- Experienced in both academia and industry:
  - Founded Comp.Sci. Dept.; served as Professor and Dept. Head; taught/mentored at all levels.
  - Managed/mentored programming staff for a computer reseller; self-employed as a software consultant and commercial trainer.
- At Fermilab since 1996; now in Computing Division/FPE Quadrant, specializing in C++ consulting and programming.
- Participant in the international C++ standardization process.
- Be forewarned: Based on the above training and experience, I hold some rather strong opinions about computer software and programming methodology — these opinions are not shared by all programmers, but they should be! ☺

Overview

- Programming with templates in C++:
  - Generalized parameterization and the compilation model.
  - Kinds of templates; kinds of template parameters.
  - Generic programming; customizing generic algorithms.
- Template-based metaprogramming:
  - Templates as compile-time computation engines:
    - Metafunctions and metafunction classes.
    - Type computations.
    - Compile-time control structures for selection & looping.
  - Meta-coding and -testing methodologies, and related issues.

Quick review of classical parameterization

- A function's desired behavior is traditionally expressed in terms of designated parameters (placeholders):
  - `double sqr( double x ) { return x * x; }`
- A client passes (supplies) his arguments (corresponding lvalues or rvalues) when the function is called:
  - `⋯ sqr( 3.14 ) ⋯`
- Such parameter passage:
  - Happens at run-time.
  - Initializes each parameter with its corresponding argument, as if by $T_i\ p_i\ (\ a_i\ )$; // i^{th} *parameter* p_i *of type* T_i *initialized via* a_i
  - More details presented in lecture about callability.

Generalized parameterization

- Modern C++ also provides compile-time placeholders, *e.g.*, type parameters:
  - `template< typename T >`
    `inline T sqr( T x ) { return x * x; }`
  - This lets us factor code that is the same except for the type.
- An entity such as sqr is known as a function template:
  - On demand, the compiler can instantiate (generate) a template specialization (an argument-specific version):
    - *E.g.*, `sqr<double>` *// here,* `double` *is the template argument*
  - Once instantiated, a specialization is compiled and used as an ordinary function whose name is a template-id:
    - *E.g.*, `⋯ sqr<double>( 3.14 ) ⋯`
- Templates have no other use beyond instantiation.

Specialization details

- A specialization is instantiated iff called by a client:
  - ⋯ sqr<double>(3.14) ⋯
  - This is known as implicit instantiation, and is quite common.
- A specialization can be instantiated, even if not called:
  - template sqr<long double>;
  - This is known as explicit instantiation, rarely used in practice.
- We can also define specializations of our own:
  - template< > *// alternate algorithm, just for* MyType
    double sqr<MyType>(MyType x) { ⋯ }
  - Such an explicit specialization must precede its first use.

Argument deduction

- Template arguments can often be deduced by a compiler from inspecting the type of each function argument:
  - ⋯ sqr(3.14) ⋯ // ⇨ sqr<double>(3.14)
  - MyType m = ⋯ ;
    ⋯ sqr(m) ⋯ // ⇨ sqr<MyType>(m)
- Whether explicit or deduced, each template argument participates identically in template instantiation.

The compilation model

- When compiling a call to a traditional function:
  - It suffices to provide the function's declaration.
  - The function's full definition can be provided later (and even compiled separately, if desired).
- But when calling a function instantiated from a template:
  - The compiler needs the template's full definition …
  - Because there is no function to call until the compiler instantiates one from the template.
  - Note: unlike class hierarchies, templates generate no code for functions that aren't called.
- Programmers must heed the ODR (one-definition rule):
  - Advisable to mark each function template as inline, and …
  - Fully define each function template where it's declared.

Characteristics of generic programming [D. Gregor, 2007]

- Abstraction:
  - Captures an algorithm's essence, free of irrelevancies (details specific to any particular type).
- Reusability:
  - Algorithms can work on user-defined data types.
- Composability:
  - Algorithms can work on types defined in a separate library.
- Efficiency:
  - Performance can be on a par with hand-coded (non-generic) implementations.
- Realization:
  - C++ template technology provides the machinery.

Example: std::swap< >( ) from <algorithm>

- Embodies the usual 3-step algorithm, but is coded generically (applicable to any type):
  - template< typename T >
    inline void swap(T & x, T & y) {
    T const tmp(x);
    x = y;
    y = tmp;
    }
- Note reliance on T's concepts (properties/interface):
  - *E.g.*, in the above, T must be copyable (have a copy c'tor as well as a copy assignment operator).
  - Several compile-time enforcement (concept-checking) techniques are known and available today.

The future of concepts in C++

- The next C++ standard ("C++0x") provides support for constrained genericity, incorporating:
  - Syntax to express requirements as named concepts, and …
  - A concept map mechanism by which a given entity is defined/inferred to satisfy a given concept.
- A library of standard concepts is provided:
  - #include <concepts>
    template< std::Copyable T > // swap's T *is now constrained*
    inline void swap(T & x, T & y) { ⋯ }
- Programmers may also define concepts of their own:
  - Either from first principles, or …
  - Via refinement (in terms of prior concepts).

Generic programming pitfalls 1

- Extra thought is often necessary when designing generic components:
  - template< typename X, typename Y >
    void swap(X & x, Y & y) { *// No!*
      X const tmp(x);
      x = y;
      y = tmp;
    }
- Above relies on X ⇔ Y conversion concepts:
  - Post-swap( ) values need not match original values, …
  - Hence swap( )-ing twice can't always restore originals.
  - Such counter-intuitive behavior ought be avoided.

Generic programming pitfalls 2

- Extra thought is often necessary when coding generic components:
  - template< typename InIter, typename T >
    InIter find(InIter first, InIter last, T const & val) {
      while ((first < last) && ! (* first == val))
        ++ first;
      return first;
    }
- This code works when called with std::vector< > iterators, but fails when called with, *e.g.*, std::list< > iterators:
  - Input iterators need not provide operator < ( ).
  - vector< > iterators do, but list< > iterators don't!
  - find< >'s author should have written (first != last).

Class templates

- Example, excerpted from <complex>:
  - template< typename T >
    class complex {
    public:
      complex(T r, T i) : real_part(r), imag_part(i) { }
      T real( ) const { return real_part; }
      // …
    private:
      T real_part, imag_part;
    };
- Usage: std::complex<double> c (0.0, 3.14);
- While std::complex names a class template, the template-id std::complex<double> names a type.

Class template details

- Each function member is a function template:
  - Each is independently instantiated, iff needed.
  - Its template parameters are those of the class template.
  - Each may also have additional template parameters of its own (a member function template).
- Recommended practice is to publish each of a class template's arguments:
  - template< typename T >
    class Container {
    public:
      typedef T value_type;
      // …
    };

Why publish template arguments?

- Consider a generic algorithm over a container type C:
  - template< typename C >
    what_return_type?
      add_em_up(C const &);
- Since we want this algorithm's return type to match the container's element type:
  - We ask the container type for the (published) type of its elements.
  - template< typename C >
    typename C::value_type
      add_em_up(C const &);

Unlike function templates …

- Class template arguments must be provided explicitly:
  - Why? No function call, so no function arg's for deduction.
  - complex<long double> c;
- Class template trailing parameters may have defaults:
  - template< class T, class U = int >
    class MyType { … };
  - Extended to function template parameter defaults in C++0X.
- Class templates may be partially specialized:
  - template< class U >
    class MyType<bool, U> { … };
  - Function templates can achieve the equivalent by calling a function object of a type that has been specialized.

Writing and using specializations

- First define a primary template for the general case:
  - template< class T >
    class C { */* handle most types* T *this way */* };
- Then define special cases, if any:
  - template< class T >
    class C<T *> { */* handle all pointer types this special way */* };
  - template< >
    class C<int *> { */* treat pointers-to-*int *even more specially */* };
- The compiler will:
  ① Select the most applicable primary class template, then …
  ② Select or instantiate the most appropriate specialization of that primary template.

Function objects in (very) brief

- An object whose type has an operator ( ) member is a function object.
- Function objects are similar to functions in many respects:
  - void func(float x) { … }
  - class FuncObj { …
    public:
    void operator ( ) (float x) const { … }
    };
  - Each is callable via ordinary function-call syntax:
    - But a function object, as an otherwise ordinary variable, must be instantiated before it can be called.
    - Can instantiate, then call right away: FuncObj( ) (3.14F).
- More details presented in lecture about function objects.

Generic algorithms customized via user plug-ins

- A generic algorithm can call a function or a function object:
  - Only the types are different; the calling syntax is the same …
  - So a function template can be used to factor this distinction.
  - template< typename CallableEntity >
    void algo(CallableEntity f, float q) *// f is a callback*
    { f(q); }
- A call to such a generic algorithm can then supply:
  - A function:
    - algo(func, 3.14F); *// same as passing* & func
  - Or a function object:
    - FuncObj my_f ;
      algo(my_f, 3.14F); *// using a named object*
    - algo(FuncObj( ), 3.14F); *// using an anonymous object*

Template template parameters

- A template parameter can itself resemble a class template:
  - template< template<class> class Bag >
    class C1 { …
    Bag<float> b;
    };
- Such a template template parameter can be used in conjunction with other template parameters:
  - template< class E, template<class> class Bag >
    class C2 { …
    Bag<E> b;
    };
- Experience has shown that template template parameters are best avoided whenever possible, so we'll say no more.

Non-type template parameters

- Non-type template parameters encompass values of:
  - Integral types: int, long, long long, bool, enum, …
  - Pointer types: int *, std::string *, void (*)(int), …
- Example:
  - template< unsigned N >
    inline double pow(double x) { *// calculate* x^N; version 1
    double result(1.0);
    for (unsigned k = 0 ; k != N ; ++ k)
    result *= x;
    return result;
    }
    The template argument's value must be known at compile time.
  - Usage: pow<3>(3.14159) approximates π^3.
  - Time complexity: O(N) (count the multiplications).

An improved algorithm

- Based on the mathematical identity $x^N = (x^2)^{N/2}$:
  - Uses only the minimum number of multiplications needed …
  - Thus reducing the time complexity to $O(\log_2 N)$.
  - *E.g.*, for $x^8 = x^{2^2}$, v1 uses 8 mult's, but v2 uses only 3 mult's.
- Implementation accounts for integer-division-by-2 chop:
  - template< unsigned N >
    inline double pow(double x) { *// calculate* x^N; version 2
    return (N % 2u) ? x * pow<N / 2u>(x * x) *// odd* N
    : pow<N / 2u>(x * x) ; *// even* N
    }
  - template< > *// specialization handles the base case*
    inline double pow<0u>(double x) { return 1.0; }

Representative timings (2001)

	std::pow( )	pow<>( ) v1	pow<>( ) v2
real	11.858 s	8.081 s	3.035 s
user	11.837 s	8.081 s	3.024 s
sys	0.020 s	0.020 s	0.030 s

- Measured x^{50} repeated 10,000,000 times.
- Used gcc 2.95.2 on 700 MHz PIII, Win2K.

Timings with optimization (2001)

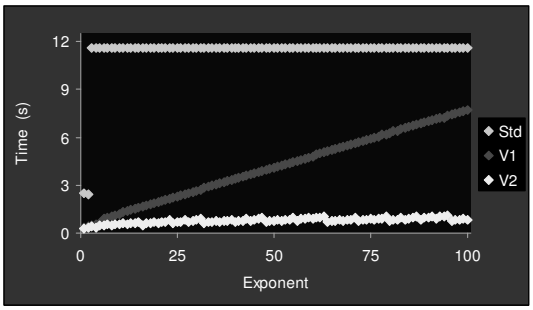
	std::pow( )	pow<>( ) v1	pow<>( ) v2
real	11.857 s	4.286 s	0.300 s
user	11.847 s	4.236 s	0.190 s
sys	0.020 s	0.010 s	0.010 s

- Used *g++ -O2* to compile; otherwise identical.
- Improvements: ~47% for v1; ~90% for v2!

Why such a dramatic decrease?

- "[Use of] templates will often result in code that is orders of magnitude faster (because costs are pushed to compile time)."
  — Francis Glassborow
- "Most of the win … comes from the additional optimization possibilities that are opened up."
  — James Kanze

Performance for increasing powers



Generic algorithms customized via traits

- A common scenario in generic programming:
  - Generic algorithm G needs some minor adjustments that vary according to its type argument U.
  - Could specialize for each G<U>, but this is burdensome on the user and defeats the point of using generic code.
  - Ideally would like to have G routinely ask U for guidance, preferably at compile-time.
  - Since type U may be built-in, user-defined, or from the standard library, there's no single interface to rely on.
- Solution: invent a specializable traits type for G to consult.

Overview of traits' use

- G's author first provides a primary template GTraits< >:
  - Often termed traits/policies/strategy/behavior template.
  - Serves as an intermediary between G and the user's type U.
- Each GTraits<U> (primary and specialization) consists of:
  - Interfaces to U's needed functionality or property …
  - In the form of ① typedefs and/or ② static members.
- G's author ensures that G<U> will use GTraits<U> for any needed adjustments/customizations:
  - G uses the primary GTraits<U> by default, but …
  - G's user can specialize GTraits<U> to handle U's quirks.

Skeleton of traits/policies in action

- template< class > struct Gtraits { *// provide default traits for* G
  ⋯ *// define members* return_type *and* static s( )
  };
- template< class U, class Tr = Gtraits<U> >
  class G { *// generic algorithm using traits*
  typedef typename Tr::return_type type;
  public:
  inline type operator ( ) ( ) { return Tr::s( ); }
  };
- template< > struct Gtraits<MyT> { *// specialized traits*
  typedef double return_type;
  static inline return_type s( ) { return 3.14; }
  };
- ⋯ G<MyT>( ) ( ) ⋯ *// user application code*

Templates in brief

- Summary so far:
  - Function templates and class templates.
  - Type, non-type, and template template parameters.
  - Generic programming and its customization via:
    - Functions/function object callbacks (run-time).
    - Traits/policies (compile-time).
- Significant inherent computational power:
  - Realized only long after templates' initial design:
    - Explains unfamiliar, somewhat awkward syntax.
    - Is surprisingly powerful; Turing-completeness claimed.
  - Exploited via template metaprogramming: advanced compile-time computation.

Selected Topics in Computer Programming #3

C++ Templates and Template Metaprogramming

End of Part 1

Part 2 will discuss advanced techniques of template-based compile-time computation, including compile-time control structures.

Overview

- ✓ C++ programming with templates:
  - ✓ Generalized parameterization and the compilation model.
  - ✓ Kinds of templates; kinds of template parameters.
  - ✓ Generic programming; customizing generic algorithms.
- Template-based metaprogramming:
  - Templates as compile-time computation engines:
    - Metafunctions and metafunction classes.
    - Type computations.
    - Compile-time control structures for selection & looping.
  - Meta-coding and -testing methodologies, and related issues.

Compile-time computation

- Template metaprogramming applies template instantiation as a *compile-time evaluation* mechanism.
- Example: compile-time absolute value metafunction:
  - template< int N >
    struct abs {
    // publish the computed value:
    static int const value = (N < 0) ? – N : N;
    // sanity check (C++0X):
    static_assert(value >= 0, "abs: overflow!");
    };
- Usage (a metafunction call):
  - int const myNum = ⋯ ;
    ⋯ abs<myNum>::value ⋯ *// yields a compile-time constant*

Combining kinds of parameters

- *E.g.*, compile-time rank of an array type (from <type_traits>):
  - template< class T >
    struct rank { *// primary template: non-array case*
    static size_t const value = 0u ;
    };
  - template< class T, size_t N >
    struct rank<T[N]> { *// specialization: array case*
    static size_t const value = 1u + rank<T>::value ;
    };
- Usage:
  - typedef sometype array_t [10] [20] [30];
    ⋯ rank<array_t>::value ⋯ *// yields 3 (at compile-time)*

Making compile-time decisions

- An instantiation that yields a type is a type computation:
  - *E.g.*, imagine a metafunction that selects one of two types.
  - template< bool, class, class >
    struct IF { typedef ⋯ type ; }; *// publish the resulting type*
- Such a facility lets us write self-configuring code:
  - *E.g.*, int const q = ⋯ ; *// user's configuration parameter*
  - Now we let the compiler make decisions based on q's value:
    - IF< (q<0), int, unsigned >::type k; *// declare* k *as 1 of 2 types*
    - IF< (q<0), F, G >::type( )(⋯) *// call 1 of 2 function objects*
    - class D : *// inherit from 1 of 2 base classes*
      public IF< (q<0), B1, B2 >::type { ⋯ };
  - Can thus be considered a form of compile-time control flow.

IF< > behind the scenes

- Straightforward to implement via partial specialization:
  - template< bool p, class T, class F >
    struct IF *// general case: assumes* true *predicate* p
    { typedef T type ; };
  - template< class T, class F >
    struct IF<false, T, F> *// special case: false predicate*
    { typedef F type ; };
- Alternate (long-winded) implementations are available for older, broken compilers, but are rarely needed nowadays.
- Such an IF<>, named std::conditional<>, is in the new C++0X header <type_traits>.

Combining several techniques

- Compile-time gcd (greatest common divisor) metafunction:
  - Euclid's algorithm (version 1) uses two helper metafunctions.
  - template< int M, int N >
    struct gcd {
    private:
    typedef typename conditional< (N == 0)
    , gcdLast<M,0> *// final step*
    , gcdNext<M,N> *// next step*
    >::type
    Step;
    public:
    static int const value = Step::value;
    };

gcd< >'s helpers

- One helper takes the next step of Euclid's algorithm:
  - template< int M, int N >
    struct gcdNext { *// invoked from* gcd<M,N>
    static int const value = gcd<N, M % N>::value;
    };
- The other helper interprets the result when done:
  - template< int M, int >
    struct gcdLast { *// invoked from* gcd<M,0>
    static int const value = M;
    };
  - template< >
    struct gcdLast<0,0> { *// invoked from* gcd<0,0>
    static int const value = 1;
    };

gcd< > v2: one fewer helper

- Absorb gcdNext<> into main template:
  - template< int M, int N >
    struct gcd {
    private:
    typedef typename IF< (N == 0)
    , gcdLast<M,N> *// final step*
    , gcd<N, M % N > *// next step*
    >::type
    Step;
    public:
    static int const value = Step::value;
    };

gcd< > v3: specializations as helpers

- Primary template:
  - template< int M, int N >
    struct gcd {
    static int const value = gcd<N, M%N>::value;
    };
- A partial specialization produces gcd(m, 0):
  - template< int M >
    struct gcd<M,0> { static int const value = M; };
- A complete specialization produces gcd(0,0):
  - template< >
    struct gcd<0,0> { static int const value = 1; };

Compile-time callbacks via metafunction classes

- We want the ability to pass metafunctions as arguments to other metafunctions:
  - But this would require routine use of template template parameters, which we strongly prefer to avoid.
- Instead we package the metafunction in the form of a member template:
  - Conventionally named apply, …
  - Taking suitable template parameters, and …
  - Wrapped in a class named for the function's purpose.
- This technique allows the wrapper class to be passed as an ordinary template type argument.

gcd<> repackaged into metafunction class Gcd<>

- struct Gcd {
  template< int M, int N >
  struct apply {
  static int const value = gcd< N, M >::value;
  };
  };
- Such a metafunction class is easily passed as an argument to another metafunction class:
  - ⋯ F::apply<Gcd> ⋯
- A metafunction class is still easy to invoke:
  - ⋯ Gcd::apply<x,y>::value ⋯

A useful variation on conditional< >

- Provides no nested type for a false predicate:
  - template< bool p, class T = void >
    struct enable_if *// general case: true predicate* p
    { typedef T type ; };
  - template< class T >
    struct enable_if<false, T> *// special case: false predicate*
    { */* no nested* type*! */* };
- Now consider enable_if< false, MyType >::type:
  - Always an error, right?
  - No, only sometimes an error: SFINAE!
  - SFINAE: Substitution Failure Is Not An Error.

SFINAE during overload resolution

- If several functions or function templates share the same name (*e.g.*, f) in the same scope, the name is overloaded.
- Each call to such an f requires overload resolution (a compiler process to decide which f best matches the call):
  ① Form a list of only viable candidates (*e.g.*, excluding f's having the wrong number of parameters, *etc.*).
  ② Rank those viable candidates (generally preferring fewer and simpler argument conversions).
  ③ Select the unique best match; if none, the call is ambiguous.
- A template is not a candidate, but an instantiation may be:
  - So overload resolution induces template instantiation(s).
  - Here, ill-formed instantiations are neither viable nor in error.

SFINAE in use

- *E.g.*, want to use one generic algorithm for integral types T, and a different generic algorithm for floating-point types T.
- For a given type T, allow at most one of the two algorithms to be instantiated:
  - template< class T >
    typename enable_if< is_integral<T>::value
    , std::maxint_t
    >::type
    f (T val) { ⋯ };
  - template< class T >
    typename enable_if< is_floating_point<T>::value
    , long double
    >::type
    f (T val) { ⋯ };

Even compile-time iteration is possible

- Design based on the C++ run-time while statement.
- Semantics based on 3 user-supplied parts:
  - Initialization: a distinguished start state.
  - Condition: a state → bool metafunction.
  - Body: a state → state metafunction.
- A compile-time WHILE maps such a triple to a (final) state:
  - template< class Cond, class Body >
    struct WHILE { *// simplified*
    template< class State >
    struct apply { typedef ⋯ type; };
    };

Use as part of larger calculation

- Calculate n! at compile-time:
  - ```
    template< int N >
    struct factorial {
    private:
      typedef typename WHILE< Cond, Body
                              >::apply< State<N> >::type
              FinalLoopState;
    public:
      static long const value = FinalLoopState::prod;
    };
    ```
- Usage:
  - ⋯ factorial<10>::value ⋯ *// now a compile-time constant!*

How to test?

- First thought:
  - ```
    std::cout << "0! is " << Factorial< 0 >::value << '\n';
    std::cout << "1! is " << Factorial< 1 >::value << '\n';
    std::cout << "2! is " << Factorial< 2 >::value << '\n';
    std::cout << "3! is " << Factorial< 3 >::value << '\n';
    std::cout << "4! is " << Factorial< 4 >::value << '\n';
    ```
- Observation: this is repetitive. (Obviously!)
- Would prefer to automate its generation:
  - TestFactorial<5>( ); *// this works!*

Writing metaprograms

- Most of us have little experience with this style of programming:
  - The programming concepts are unfamiliar.
  - Their utility is unfamiliar.
  - The underlying thought processes are unfamiliar.
- So what are some of the trade-offs?

Metaprogramming's memory usage

- Code space:
  - Function templates are instantiated only if called.
  - Use of conditional<> can reduce necessary instantiations.
  - Code optimization opportunities are plentiful.
- Data space:
  - Analogous to 1+2: the object code has only a 3.
  - Metafunctions, even if instantiated, rarely leave any memory footprint for data.
- "Code bloat" can be an artifact of inexperience, or of limitations in compiler technology.

Metaprogramming's use of CPU time

- Build time may increase somewhat:
  - Compiler is doing (considerably) more work.
  - Compile time often supplants execution time.
  - Usually a good tradeoff, especially in production code.
- Build time may decrease somewhat:
  - Unused code is not compiled.
  - Better inlining decreases link time.

Overall metaprogramming efficiency

- Many programmers confuse efficiency with elapsed-time performance:
  - A program giving the wrong answer at the speed of light is fast, but certainly is not efficient!
  - Efficiency is correctly measured as a weighted sum of all relevant costs.
- Yes, metaprogramming can certainly be an important source of improved performance:
  - Unfamiliar, so still has a longish learning curve for now.
  - Is rapidly becoming increasingly important in modern C++ programming.

Recent developments

- Programming with C++ templates has become increasingly familiar to more programmers.
- C++ template-based metaprogramming has become well-understood by experts.
- Very good tutorial materials have been produced, where once there were none.
- Important abstractions have been identified.
- These abstractions have been encapsulated in libraries, for use by non-experts.

For more information

- C++ language:
  - Koenig & Moo: Accelerated C++, 2000.
  - Stroustrup: The C++ Programming Language, 2000.
- C++ programming technique:
  - Sutter & Alexandrescu: C++ Coding Standards, 2005.
  - Dewhurst: C++ Gotchas, 2003.
- C++ templates and metaprogramming:
  - Vandevoorde & Josuttis: C++ Templates, 2003.
  - Abrahams: "Generic Programming Techniques," 2004.
  - Alexandrescu: Modern C++ Design, 2001.
- C++0X concepts:
  - Gregor: http://video.google.com/videoplay?docid=-1790714981047186825, 2007.

FIN

Selected Topics in Computer Programming #3

C++ Templates and Template Metaprogramming

Walter E. Brown, Ph.D. <wb@fnal.gov>
Computing Division
Fermi National Accelerator Laboratory