CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

C++ Course

Generic Programming Part I : Templates (+ some other stuff thrown in)

John Biddiscombe / Mauro Bianco

Generic programming

Wikipedia: "Generic programming is a style of computer programming in which algorithms are written in terms of data types to-be-specified-later that are then instantiated when needed for specific types provided as parameters."

Us:

- Code re-use
  - (Design) Patterns
- Abstraction of algorithms and data
  - Look no further than the STL (it *is* amazing)
- API
  - Standardize the API for algorithms/operations
- Use of operators and predicates
  - Supplying functions to algorithms (as well as data)

Functions and Classes (a reminder)

- A function is defined as return type, name, and arguments
  - Functions *exist* as code that can be linked to and called/inlined

```
int add(int x, int y) { return x + y; }
int main() { add(65, 35); }
```

- A class is defined as a name after class (or struct)
  - Definition contains type, data and function members

```
class name1;  // by default members are private
struct name2; //                         public

class name1 {
    // insert more interesting things here ...
    int thing1;
    int func2(float) { ... };
};
int main() {
  name1 x; // here we instantiate a variable of type 'name1'
}
```

Function templates - Instantiation

- A function template is not a function
  - It needs to be *instantiated*
  - Substitute type text into the template argument
  - Trivial link

```
template <typename T>
void foo(T x) {
    std::cout << x << "\n";
}

int main() {
    foo<int>(65);
    foo<char>(65);
    foo<double>(3.1415);
    foo<std::string>(std::string("string"));
}
```

```
    auto *p1 = &foo;       // No! this doesn't exist
    auto *p2 = &foo<int>; // Yes - address of a specific instantiation
```

Regular Function Overloading - Note

- Overloading functions is quite normal, regardless of templates
- It's a form of specialization/customization of the function itself
- Note: Functions can't be overloaded based on return type alone
  - introduces ambiguities (+ problems with return type conversions)

```
    void function1(int x);
    void function1(float x);
    void function1(int x, float y, std::string z);
    int  function1(int x) {}                          // No!
```

```
error: functions that differ only in their return type cannot be overloaded
```

```
    struct thing1 {
        thing2 function1(int x);
        const thing2 function1(int x) const;
    };
```

- a const modifier on member function changes the function signature (type)
  - so the *change* in return type is ok

Template Function Overloading - Deduction

- Among the options the most specialized is chosen
  - Introduce the term "ADL" - Argument Dependent Lookup (More later)

```cpp
template <typename T>
void foo(T x) {
    std::cout << x << ", " << typeid(x).name() << "\n";
}

void foo(std::string const& x) {
    std::cout << "ooh! a string! " << x << "\n";
}

int main() {
    foo<int>(65);                // explicit (int)
    foo<char>(65);               // explicit (char)
    foo(3.14159);                // Argument Deduction (double)
    foo<double>(3.14159);        // explicit (double)
    foo<double>(3);              // explicit (double - automatic type conversion)
    foo(std::string("yay \o/")); // Argument Deduction (string)
}
```

```
65, i
A, c
3.14159, d
3.14159, d
3, d
ooh! a string! yay \o/
```

Order Matters

- Be careful when mixing explicit and deduced arguments
- explicit types must appear where expected otherwise substitution fails

```cpp
template <typename T, typename U>
void foo(T, U) {}

int main() {
    foo<std::string, double>("hello", 4.5); // ok - string double fits
    foo<std::string>("hello", 4.5);         // ok - string came first
    foo<double>("hello", 4.5);              // not ok, can't deduce "std::string" in 1st place
}
```

```
cannot convert '"hello"' (type 'const char [6]') to type 'double'
```

- Deduction of types doesn't allow for arbitrary rearranging of template positions
- Always watch out for automatic type conversions `const char [6]` -> `std::string`

Template Argument Deduction

- To instantiate a template **all the types must be known**
- Sometimes/Usually/Often/Mostly they can be deduced

```
template <typename To, typename From>
To convert1(From f) {
    std::cout << typeid(To).name() << " "
              << typeid(From).name() << "\n";
    return static_cast<To>(f);
}

void g1(double d) {
    // To = int,  deduced From = double
    int i = convert1<int>(d);
    // To = char, deduced From = double
    char c = convert1<char>(d);
    // deduced To = int,  deduced From = float
    int(*ptr)(float) = convert1;
    ptr(d);
}
```

```
i d  c d  i f
```

[Link to example]

```
template <typename From, typename To>
To convert2(From f) {
    std::cout << typeid(To).name() << " "
              << typeid(From).name() << "\n";
    return static_cast<To>(f);
}

void g2(double d) {
    // To = int,  From = double
    int i = convert2<double, int>(d);
    // To = char, From = double
    char c = convert2<double, char>(d);
    // To = char,  From = float
    int(*ptr)(float) = convert2;
    ptr(d);
}
```

```
i d  c d  i f
```

Note : Automatic Type Conversion and Deduction

```
struct thing1 {
    double some_value = 1.0;
    operator bool() const { return some_value > 0.1; }
};
template <typename T>
void print_thing(const T &t) {
    std::cout << "Overload T " << t << " (" << t.some_value << ")" << "\n";
}
void print_thing(bool t) {
    std::cout << "Overload B " << std::boolalpha << t << "\n";
}
int main() {
    thing1 t1{0.05};
    print_thing(t1);                    // uses template - no overload for type thing1
    print_thing(static_cast<bool>(t1)); // uses overload - specialization for bool
}
```

`Overload T 0 (0.05)` `Overload B false`

- The `<<` operator picks the bool conversion, but the template doesn't (more specialized)

Function Lookup Based on Argument (ADL) #1 a)

- Which version of a function should be used when there's a choice
  - Argument Dependent Lookup. In this case, the argument types

```
struct thing1 {
    double some_value = 1.0;
    operator bool() const { return some_value > 0.1; }
    friend std::ostream & operator << (std::ostream &os, const thing1 &t) {
        os << t.some_value; return os;
    }
};

int main() {
    thing1 t1{3.13};
    std::cout << t1 << " " << static_cast<bool>(t1) << std::endl;
}
```

- The call to `std::cout << t1` is essentially translated to `operator << (ostream, thing1)`

Function Lookup Based on Argument (ADL) #1 b)

- We can move the stream operator out of the struct if we want

```
struct thing1 {
    double some_value = 1.0;
    operator bool() const { return some_value > 0.1; }
};

std::ostream & operator << (std::ostream &os, const thing1 &t) {
    os << t.some_value; return os;
}

int main() {
    thing1 t1{3.13};
    std::cout << t1 << " " << static_cast<bool>(t1) << std::endl;
}
```

- We must drop the friend keyword, But ...
  - what happens if we change `struct` to `class`

    `'double thing1::some_value' is private within this context`

ADL - Namespace usage

- The namespace of `t1`, `t2` are searched to find the right function
  - just like when you call `std::cout << std::string('stuff')`

```
namespace one {
    struct thing1 {
        double some_value = 1.0;
        operator bool() const { return some_value > 0.1; }
    };

    template <typename T> void print_thing(const T &t) {
        std::cout << "one Overload T " << t
                  << " (" << t.some_value << ")" << "\n";
    }
}
```

```
namespace two {
    struct thing1 {
        double some_value = 1.0;
        operator bool() const { return some_value > 0.1; }
    };

    template <typename T> void print_thing(const T &t) {
        std::cout << "two Overload T " << t
                  << " (" << t.some_value << ")" << "\n";
    }
}
```

```
int main() {
    one::thing1 t1{0.05};
    two::thing1 t2{0.07};
    print_thing(t1);
    print_thing(t2);
}
```

`one Overload T 0 (0.05)` `two Overload T 0 (0.07)`

Link to longer (qualified) version

Class templates

- A class template is not a class
  - It needs to be instantiated, only then can an object of that type exist

```
template <typename T>
class thing {
    using type = T;
    T member;
    std::unique_ptr<T> other;
    T operator()(T a, T b) const {...}
};
int main() {
  thing<int> x;
  thing<double> y;
}
```

- The template parameter may be used in any of the members or functions declared in the class
  - Function return type
  - one or more parameters used in a member function
  - used in a typedef, or declaration of another type

Partial specialization

- As mentioned - for template functions - these are really just **overloads**
- The more specialized version is chosen when encountered

```
template <typename T, typename U>
struct X {};                          // 1 (Primary template)

template <typename T>
struct X<T, int> {};                  // 2 (Specialization of arg2 / U)

template <typename U>
struct X<float, U> {};                // 3 (Specialization of arg1 / T)

int main() {
    X<char,  double> a; // choose 1
    X<char,  int>    b; // choose 2
    X<float, double> c; // choose 3
    X<float, int>    d; // all 3 match. what to do ???
}
```

```
error: ambiguous partial specializations of 'X<float, int>'
```

Full specialization

```
template <typename T, typename U>
struct X {};                          // 1 (Primary template)

template <typename T>
struct X<T, int> {};                  // 2 (Specialization of arg2 / U)

template <typename U>
struct X<float, U> {};                // 3 (Specialization of arg1 / T)

template <>                           // this "template <>" is important
struct X<float, int> {};              // 4 (Full Specialization)

int main() {
    X<char,  double> a; // choose 1
    X<char,  int>    b; // choose 2
    X<float, double> c; // choose 3
    X<float, int>    d; // choose 4
}
```

- Full specialization is **more** specialized, so it is picked and resolves the problem

Pattern Matching #1

```
template <typename T, typename U>
struct X {};                            // Primary

template <typename W, typename T, typename U>
struct X<W, X<T,U>> {};                 // Specialization 1

template <typename T>
void foo(X<T,T>) {}                     // Function 1

int main() {
    X<int, X<int, float>> a;            // specialization<int, primary>
    X<int, X<char, X<int, void>>> b;    // specialization<int, primary>
    X<double, double> c;                // primary
    foo(c);                             // ok calls Function 1
    foo(b);                             // no :( does not match
}
```

```
candidate template ignored: deduced conflicting types for parameter 'T'
('int' vs. 'X<char, X<int, void>>') -> void foo(X<T,T>) {}
```

Pattern Matching #2

```
template <typename T, typename U>
struct X {};                            // Primary

template <typename W, typename T, typename U>
struct X<W, X<T,U>> {};                 // Specialization 1

template <typename T>
void foo(X<T,T>) {}                     // Function 1

template <typename T, typename U>
void foo(X<T,U>) {}                     // Function 2

int main() {
    X<int, X<int, float>> a;            // specialization<int, primary>
    X<int, X<char, X<int, void>>> b;    // specialization<int, primary>
    X<double, double> c;
    foo(c);                             // ok F1
    foo(b);                             // yay \o/ F2
}
```

- Now the `U` parameter is deduced to be the full `X<char, X<int, void>>`

CRTP (Curiously Recurring Template Pattern) : static polymorphism

```
template<typename Derived>
class controller_base {
  template <typename... Args>
  bool initialize(int i, bool b, Args... args) {
      // lots of boilerplate code here
      return static_cast<Derived*>(this)->initialize_derived(i, b, std::forward<Args>(args)...);
  }
}
class controller : public controller_base<controller> {
  void initialize_derived(int i, bool b, std::string const& s) {
      // some special init for this type of controller
  }
}

int main() {
    controller c;
    c.initialize(42, true, "some stuff");
}
```

- Polymorphism allows a call to be directed to the intended type
  - shape->circle/square/blob - inheritance - traditionally uses VMT lookup
- CRTP creates an "interface" that can be customized by derived types
  - routing made at compile time - *because types are already known*

Default template arguments

- Template arguments can be defaulted
  - From C++11 this is also possible on function templates
- Default args can only be to the right the arguments on the right **if not deduced**

```
template <typename T, typename Result=char>
Result foo(T x) {
    return static_cast<Result>(x);
}

int main() {
    std::cout << foo(65) << "\n";             // T deduced to be int, Result char
    std::cout << foo<int>(65) << "\n";        // T explicitly int,    Result char
    std::cout << foo<int, int>(65.3) << "\n"; // T and Result, both explicitly int
}
```

```
A A 65
```

- Note that because we used `static_cast<Result>(x)` the `double` **65.3** was converted to `int` without error

Default args go on the right *unless they can be deduced*

```
template <typename T, typename Result=char, typename Another>
Result foo(T x, Another a) {
    std::cout << typeid(T).name() << " " << typeid(Another).name() <<" ";
    return static_cast<Result>(x);
}

int main() {
    // T deduced as int, Another deduced as double
    std::cout << foo(65, 3.5) << "\n";                    // 1 Result = char
    // but like this we explicitly state
    std::cout << foo<int, char, float>(65, 3.5) << "\n";  // 2 Result = char
    // in this case Another is deduced as double
    std::cout << foo<int, float>(65, 3.5) << "\n";        // 3 Result = float (careful)
}
```

output : `i d A` `i f A` `i d 65`

1. The compiler knows `T` and `Another` so it is ok with defaulting `Result`
2. We tell the compiler all 3
3. The compiler can deduce `Another` but we forced `Result` to be float

Template Templates : More on deduction

- Don't try to put the `V` template inside the `U` definition

```
template <typename T, template <typename> typename U, typename V>
void foo(T x, U<V> a) {
    std::cout << "1 " << typeid(T).name() << " " << typeid(U<V>).name() <<"\n";
}

template <typename T, template <typename> typename U>
void foo(T x, U<T> a) {
  std::cout << "2 " << typeid(T).name() << " " << typeid(U<T>).name() <<"\n";
}

int main() {
    std::vector<double> vect;
    std::list<float> vecl;

    foo(4.5, vect);
    foo(4.5, vecl);
}
```

`2 d St6vectorIdSaIdEE` `1 d NSt7__cxx114listIfSaIfEEE`

Link to example

Non type template arguments

- Drop the `typename` syntax and instead specify a concrete `type`
- Values can be used as template arguments
  - bool, char, int, pointers and arbitrary types
- Non type templates can also have default values

```
template <int I>
struct static_int {
    // a data value hardcoded for this type
    static constexpr int value = I;
};

int main() {
    std::cout << static_int<5>::value;
    std::cout << static_int<6>::value;
}
```

- Be aware that `static_int<5>` and `static_int<6>` are not the same *type*
- You could (for example) specialize another template using `static_int<XXX>` types

Default Template arguments for classes

```
template <typename T=double, int Size=10>
class my_container {};

int main() {
    // x is a my_container of 10 doubles
    // <> required because container is templated and needs default types
    my_container<> x;

    // how does the compiler know what type the elements are?
    my_container<100> y; // ERROR

    // STL uses this approach for std::array (replaces double[N] syntax from C)
    std::array<double, Size> lovely;

    // Nicer than the old way (no size info carried)
    double deprecated[Size]
}
```

- Constructions like `std::array<char, Size>` are very useful for things like
  - message buffers/headers
  - cache line padding
  - Small buffer Optimization

Evaluation order

- Template arguments are evaluated left to right
  - So we can extract types from the first and use them in the second, etc ...
- C++ gets interesting and powerful when you allow types to expose other types
  - or (other) embedded types that have been specialized ... link

```
template <typename T, typename U = typename T::type, int X = U::value>
struct Order {};

struct B {
    using type = B;
    static const int value = 100;
};

struct B2 {
    struct C {
        static const int value = 42;
    };
    using type = B2::C;
};

int main() {
    Order<B> x;
    Order<B2> y;
}
```

Evaluation order (wrong)

- You can't use a derived type/value before it is defined
- Swapping U/X breaks the compilation

```
template <typename T, int X = U::value, typename U = typename T::type>
struct Order {};

struct B {
    using type = B;
    static const int value = 100;
};

int main() {
    Order<B> x;
}
```

```
error: 'U' has not been declared
```

```
error: template argument 2 is invalid
```

Alias templates

- typedefs on steroids!
  - But they are still really just typedefs

```
using integer_type = int;
// same as
typedef int integer_type;
```

- But you can template them, which is really helpful

```
// create an alias for a std::vector
template <typename T>
using my_type = std::vector<T>;

// and then later on
my_type<double> x(100);
```

- Many template arguments and defaults are allowed

```
// create an alias for an STL-like container (which is itself templated)
template <template <typename, typename> typename T, typename U,
          template <typename> typename Alloc = std::allocator>
using my_container = T<U, Alloc<U>>;

my_container<std::vector, int> x(100);

// you can fill in (for example) an allocator to save the user doing it
std::vector<int, std::allocator<int>>
```

Alias templates (it's an alias, not a new type)

```
template <typename T>
using my_vec = std::vector<T, my_allocator<T>>;

template <typename T> // defintion 1
std::size_t size_of(my_vec<T> const& v) { return v.size(); }

template <typename T> // defintion 2 - this is the same as 1
std::size_t size_of(std::vector<T, my_allocator<T>> const& v) {
    return v.size();
}
```

```
error: redefinition of 'size_of' std::size_t size_of(std::vector<T, my_allocator<T>> const& v)
```

Alias templates (it's an alias, not a new type)

```
template <template <typename, typename> class V>  // 3
std::size_t size_of(V<int, std::allocator<int>> const& v) { return v.size(); }

template <template <typename> class V>            // 4
std::size_t size_of(V<int> const&v) { return v.size(); }

int main() {
    std::cout << size_of(my_vec<int>(23,0)) << std::endl; // uses 3, 4 doesn't match
}
```

- even though we've aliased the vector to a single template param, it really still has 2, using #4 we get

```
template template argument has different template parameters than its corresponding template template parameter
```

Default Template Arguments and Specializations

- Which specialization applies

```
template <typename T=double, int Size=10>
struct my_container {};                    // always searched first

template <typename T>
struct my_container<T, 10> {};             // specialization 1

template <typename T>
struct my_container<T, 15> {};             // specialization 2

int main() {
    my_container<char, 10> z;              // uses specialization 1
    my_container<float,30> u;              // uses primary
    my_container<int,  15> v;              // uses specialization 2
    my_container<int>      y;              // uses specialization 1 (Size=10)
    my_container<>         x;              // primary (both default)
}
```

Specialization - Trickier selection

```
template <typename T, typename U = int>
struct X {
    X() {
        std::cout << "Primary " << typeid(T).name() << " "  << typeid(U).name() << "\n";
    }
};
template <typename T>
struct X<T, typename T::extra_type> {
    X() {
        std::cout << "Specialization " << typeid(T).name() << " " << typeid(typename T::extra_type).name() << "\n";
    }
};
struct A { using value_type = int; };
struct B { using extra_type = int; };
struct C { using extra_type = float; };
struct D { using extra_type = char; };
int main() {
    X<A> a;         // uses primary - A::value_type not a match
    X<B> b;         // uses specialization - B::extra_type = int
    X<C> c;         // uses primary - C::extra_type not int
    X<B,char> b1;   // uses primary - B::extra_type not char
    X<D,char> d;    // uses specialization, D::extra_type = char
}
```

`Primary 1A` `Specialization 1B` `Primary 1C` `Primary 1B` `Specialization 1D`

- Reducing template params from 2 down to 1
- **use specialization If `T::extra_type` matches `U`**
- Compiler explorer link : More advanced link

SFINAE

- **Substitution Failure Is Not An Error**
  - When looking for specialization some substitution may fail
    - It's the backbone of templated code
    - Without it, nothing would work (compile)

```
// some generic struct that by default will use int
template <typename T, typename U = int>
struct X {};

// a specialization for the struct *IF* it has an extra_type definition
// When extra_type is not there, clearly this would be a compilation 'fail'
template <typename T>
struct X<T, typename T::extra_type> {};
```

- When a substitution fails, the compiler ignores it and moves on
- When it succeeds, it becomes a candidate for specialization/lookup
- "concepts" and "constexpr if" can/will mostly do away with SFINAE
  - but tons of existing code still uses it

SFINAE: `std::enable_if``

```
template <typename T, typename Enable = void>
struct A;

template <typename T> // does not compile if is_same fails
struct A<T, typename std::enable_if<std::is_same<T, int>::value, void>::type> {
    A() { std::cout << "int!\n"; }
};

template <typename T> // // does not compile if is_same succeeds
struct A<T, typename std::enable_if<!std::is_same<T, int>::value, void>::type> {
    A() { std::cout << "not int\n"; }
};

int main() {
    A<int> a1;
    A<float> a2;
}
```

`int!` `not int`

- The failed version will not compile so the good version is selected - SFINAE
- The `void` template parameter is allowed to be an empty param (like void in a function)

`std::enable_if` : Possible implementations

- Example 1:
  - Specialization for true defines `type = T`
  - Specialization for false doesn't exist, so `type` doesn't either

```
template<bool B, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> {
    using type = T;
};
```

- Example 2:
  - Specialization for true defines `type = T`
  - Entire class doesn't exist for false, but SFINAE works (Not An Error)
    - We **declared** a class, but never instantiated anything for false

```
template<bool B, class T = void>
struct enable_if;

template<class T>
struct enable_if<true, T> {
    using type = T;
};
```

SFINAE: `std::enable_if_t` + `std::is_same_v` (less *cruft*)

```
template <typename T, typename Enable = void>
struct A;

template <typename T> // does not compile if is_same fails
struct A<T, std::enable_if_t<std::is_same_v<T, int>, void>> {
    A() { std::cout << "int!\n"; }
};

template <typename T> // does not compile if is_same succeeds
struct A<T, std::enable_if_t<!std::is_same_v<T, int>, void>> {
    A() { std::cout << "not int\n"; }
};

int main() {
    A<int> a1;
    A<float> a2;
}
```

- Just like the previous version, but slightly easier to understand
  - Cruft is a jargon word for anything that is left over, redundant and getting in the way. It is used particularly for defective, superseded, useless, superfluous, or dysfunctional elements in computer software

Class Template Type Deduction (C++17)

- When instantiating a templated class, the constructor can be used to deduce the type
- Caution, if A in a header and type not specified, you might not even know it's templated

```
template <typename T>
class A {
    T x;
  public:
    A(T x) : x{x} {}
};

int main() {
    A<int> x(3);    // c++ pre c++17
    A      y(3);    // A<int> is deduced via the constructor
    A      y(3.14); // A<double>  deduced
}
```

- So what?

```
template <typename F>
struct B {
    F f; // a function that we want to store and call later
    B(F&& f) : f{std::move(f)} {}

    template <typename... Args>
    void call(Args&&... args) {
        f(std::forward<Args>(args)...);
    }
};

int main() {
    // explicitly declaring the type is messy and requires temporary variable
    auto f = [](int i, int j) {cout << i+j << "\n"};
    B<decltype(f)> a{std::move(f)};

    // CTAD allows a lambda to be passed directly in
    B b{[](int i, int j) {cout << i+j << "\n";}};

    a.call(3,4);
    b.call(3,4);
}
```

- When writing algorithms that take functions/predicates and call functions in other templated parameters, knowing the exact type can become very long winded / difficult

Class Template Type Deduction (C++17)

- It still works if there are extra parameters/templates in the constructor

```
template <typename T>
class A {
    T x;
  public:
    template <typename U>
    A(T x, U, int) : x{x} {}
};

int main() {
    A<int>  x(3, 3.4, 7);
    A       y(3, 3.4, 7); // A<int>
    A       z{y};         // A<int> copy
}
```

- Internally, the compiler is doing the heavy lifting by deduction using the equivalent of auto-generated (function template) helpers

```
// Fictional function templated on <T,U> that returns the right type of A
template <typename T , typename U>
A<T> make_A(T a , U x, int y) {
    return A<T>{a, x, y};
}

// Fictional function template for a copy constructor
template <typename T>
A<T> make_A(A<T> a) {
    return A<T>{a};
}

int main() {
    A<int> x(3, 3.4, 7);          // explicit T, deduced U
    auto y = make_A(3, 3.4, 7);   // function template argument deduction
    auto z = make_A(y);           // copy usnng the same mechanism
}
```

```
std::shared_ptr ptr(new int(10)); // <sigh>
```

auto keyword

```
// forward declarations to keep compiler happy
struct thing1;
struct thing2 {
    double val{3.14};
    thing1 operator + (const thing1& t1) const;
};

struct thing1 {
    double val{3.14};
    auto operator + (const thing2& t2) const
    {
        return thing2{t2.val + val};
    }
};

thing1 thing2::operator + (const thing1& t1) const {
    return thing1{t1.val + val};
}
```

```
int main()
{
    thing1 t1{3.14};
    thing2 t2{6.28};
    auto r1 = t1 + t2;
    auto r2 = t2 + t1;
    std::cout << typeid(r1).name() << std::endl;
    std::cout << typeid(r2).name() << std::endl;
}
```

```
6thing2 6thing1
```

- When functions return unexpected or difficult to guess types (especially types derived from operations on other unknown types) - the auto keyword becomes essential

Variadic Templates

- A template parameter pack accepts zero or more arguments
  - Using `...` to express packs

```
// a function tanking zero or more arguments
template <typename... Ts> // parameter pack (the types)
void foo(Ts... args) {}   // parameter pack (the actual values)

// a class templated over zero or more types
template <typename... Ts>
class A {};
```

```
template <typename... Ts>
void foo(Ts... args) {
    function(args...);  // pack-expansion = arg1, arg2, arg3 ...
    pattern(args)...;   // pack-expansion = pattern(arg1), pattern(arg2) ...
    function(&args...); // pack-expansion = function(&arg1, &arg2, &arg3 ...);
}
```

- Whatever the `...` appears after is the thing that is being repeated

Placement of the Variadic Dots `...`

- It may at first seem confusing where to put the `...` dots
- The dots come **after** whatever is being repeated
  - Multiple typenames will be

```
template <typename... Ts>
```

  - The args types `Ts` will be a list (derived from the args)

```
void foo(Ts... args) {}
```

  - The **thing** before the `...` will be expanded to make a list

```
  pattern(args)...;
```

- :) If you see ...

```
template <typename ...Ts>
void foo(Ts ...args) {}
```

Variadics - Recursion in action

```
void pretty_print(std::ostream& s) {
    s <<"\n";
}

template <typename T, typename... Ts>
void pretty_print(std::ostream& s, T first, Ts... values) {
    s << " {" << first << "} ";
    pretty_print(s, values...);
}

int main(){
    pretty_print(std::cout, 3.2, "hello", 42, "world");
}
```

- Gives `{3.2} {hello} {42} {world}`
  Compile r Explorer link

Perfect Forwarding

- Preserves the type of the argument(s)
- const stays const,
- refs stay refs
- all modifiers are preserved when forwarding the arguments

```
template <typename... Args>
void call(Args&&... args) {
    f(std::forward<Args>(args)...);
}
```

C++17 Fold Expressions

- More elaborate pack-expansions (reductions on packs)
- 4 Flavours of Fold
  - Unary Right Fold
    - `(pack op ...)`
  - Unary Left Fold:=
    - `(... op pack)`
  - Binary Right Fold
    - `(pack op ... op init)`
  - Binary Left Fold
    - `(init op ... op pack)`
  - op may be one of the following (commonly overloaded) operators
    - `* + - * / % ^ & | = < >`
    - `<< >> += -= *= /= %= ^= &= |= <<= >>= == != <= >= && ||`
    - `, .* ->*`

Unary Left Fold example

```
template <typename... Args>
bool all(Args... args) {
    return (... && args);
}

int main()
{
    bool b = all(true, true, true, false);
    std::cout << "Result: " << std::boolalpha << b << std::endl;
}
```

- Pre-add syntax: Left fold

  `(... + vals) => (((vals1 + vals2) + vals3) + vals4)`

- Post-add syntax: Right Fold

  `(vals + ...) => (vals1 + (vals2 + (vals3 + vals4))`

Pretty print using fold

```
template<typename... Ts>
void print1(Ts... vals) {
    (std::cout << ... << vals);
}
```

- Using a binary left fold

```
template<typename... Ts>
void print2(const char *delim, Ts... vals) {
    auto showdelim = [](const char *delim, const auto& param) -> const auto& {
      std::cout << delim;
      return param;
    };
    (std::cout << ... << showdelim(delim, vals) ) << std::endl ;
}
```

- print1 does the right thing, but how do you add a delimiter?
- print2 is better, but prints an extra delimiter
- we want thr right number of delimiters and handle an empty inpput
- see fold example code for print3 link to full example

Constexpr

- The constexpr specifier declares that it is possible to evaluate the value of the function or variable at compile time.
- Such variables and functions can then be used where only compile time constant expressions are allowed (provided that appropriate function arguments are given).

```
// A literal class
class conststr
{
    const char* p;
    std::size_t sz;
public:
    template<std::size_t N>
    constexpr conststr(const char(&a)[N]): p(a), sz(N - 1) {}

    constexpr char operator[](std::size_t n) const {
        return n < sz ? p[n] : throw std::out_of_range("");
    }
    constexpr std::size_t size() const { return sz; }
};
```

```
constexpr std::size_t countlower(conststr s, std::size_t n = 0,
                                 std::size_t c = 0)
{
    return n == s.size() ? c :
        'a' <= s[n] && s[n] <= 'z' ? countlower(s, n + 1, c + 1)
                                   : countlower(s, n + 1, c);
}
```

```
std::cout << "Number lowercase letters in \"Hello, world!\" is ";
constN<countlower("Hello, world!")> out2;
```

if constexpr

- A nifty feature that allows you to get rid of some specializations

```
template<int  N>
constexpr int fibonacci() {return fibonacci<N-1>() + fibonacci<N-2>(); }
template<>
constexpr int fibonacci<1>() { return 1; }
template<>
constexpr int fibonacci<0>() { return 0; }

template<int N>
constexpr int fibonacci()
{
    if constexpr (N>=2)
        return fibonacci<N-1>() + fibonacci<N-2>();
    else
        return N;
}
```

- Much simpler on the compiler, no recursive instantiations of templates
- Code is directly evaluated at compile time

Classes and Meta-Programming

- Kinds of members
  - [Static] Function
  - [Static] Data
  - Constexpr function
  - Static const/Constexpr data
  - Type (nested type names)
- Meta-programming is manipulating types
  - And static const/constexpr values
- The main mechanism is using class templates
- Single applications rarely need TMP
- Useful when building abstraction layers
  - E.g., header-only libraries
- ODL and headers (One definition Rule)

Step Back

- Type members are possible
- Access like static members
  - `X::type_t<U>`
- Visibility rules as normal
- Constexpr variables visible at translation unit level
- inline constexpr is your friend for const vars etc

```
class X {
 /*public/private/*/protected:
    using type = ...;

    template <typename T, ...>
    using type_t = ...;

    static const int a = 10;
    static constexpr int b =10;
    inline constexpr int c = 10;

    X(...);

    void member(...);
};
```

A simple example

- A complex way to fill a register with a value

```
movl $120, %esi
```

Compiler explorer link

```
template <int N>
struct factorial {
    static constexpr int value = N * factorial<N-1>::value;
};

template <>
struct factorial<1> {
    static constexpr int value=1;
};

int main() {
    std::cout << factorial<5>::value << "\n";
}
```

- NB. `use -ftemplate-depth=N to increase recursive template instantiation depth`

A Convention for TMP

- TMP is still an “accident” in C++
  - Boost::MPL conventions partially adopted by ISO C++
- A meta-function returning a type has a public ::type
- A meta-function returning a value has a public ::value
- Usually/Frequently/Often *both*

```
template <Arguments…>
struct meta_function {
    using type = ... ;
    static constexpr ... value = ... ;
    using result_type = ... ;
    using param_type = ... ;
};
```

- Type members with arbitrary names are called traits
  - Often small helper utilities like `is_something<T>`

std::integral_constant

- A type for each number

```
template<class T, T v>
struct integral_constant {
    using value_type = T ;
    static constexpr value_type value = v;
};

// use :value to access the underlying content
static_assert(integral_constant<int, 7>::value == 7, “Error”)
```

Building abstractions: std::rank example

- Type of `T[3][4]` is `(T[3])[4]`

```
template<class T> // Primary
struct rank : public integral_constant<size_t, 0> {};

template<class T, size_t N> // Specialize for an array type
struct rank<T[N]>           // (int[3])[4] => T[4] where T = int[3]
    : public integral_constant<size_t, rank<T>::value + 1> {};

template<class T>
struct rank<T[]>
    : public integral_constant<size_t, rank<T>::value + 1> {};

int main() {
    int x[5][4][7][2];
    std::cout << rank<decltype(x)>::value << std::endl;
}
```

- Gives output of `4` Link to example

An Example with Types: If on Types

- If (boolean value) then type1, else type2

```
template <bool Pred, typename T1, typename T2>
struct select_first {
    using type = T2; // Primary (false)
};

template <typename T1, typename T2>
struct select_first<true, T1, T2> {
    using type = T1; // Specialization for true
};
```

- But why do we need this?
- One building block for more complex meta-programming functions/features

One (Maybe) Silly Example

```
template <bool WithRef>
typename select_first<WithRef, int&, int>::type
with_ref (typename select_first<WithRef, int&, int>::type x) {
    x += 1;
    return x;
}

int main() {
    int x = 1;
    std::cout << with_ref<false>(x) << " " << x << std::endl;
    std::cout << with_ref<true>(x)  << " " << x << std::endl;
}
```

- Gives `2 1` `2 2` Link
- Note use of `typename` for the return type of the function
  - the compiler needs this to warn it that the type isn't known yet
  - you see this everywhere in TMP

An example 1/6

- my_container is not a template
  - my_container is not flexible

```
struct my_container {
    using value_t = int;
    using container_t = vector<value_t>;
    container_t C;

    my_container(size_t s) : C(s) {}
};

my_container my_c(42);
```

An example 2/6

- Customizing the basics

```
template <typename VT>
struct my_container {
    using value_t = VT;
    using container_t = vector<value_t>;
    container_t C;

    my_container(size_t s) : C(s) {}
};

my_container my_c<int>(42);
```

An example 3/6

- Customizing the allocator

```
template <typename VT, typename Allocator = allocator<VT>>
struct my_container {
    using value_t = VT;
    using container_t = vector<value_t, Allocator>;
    container_t C;

    my_container(size_t s) : C(s) {}
};


my_container<int> my_c(42);
my_container<int, std::allocator<int>> my_c(42);
my_container<int, std::allocator<double>> my_c(42);
```

An example 4/6 – Template Template Arguments

- Avoiding redundancies

```
template <typename VT,
          template <typename> class Allocator = allocator>
struct my_container {
    using value_t = VT;
    using container_t = vector<value_t, Allocator<value_t>>;
    container_t C;

    my_container(size_t s) : C(s) {}
};

my_container<int> my_c(42);
my_container<int, std::allocator> my_c2(42);
```

An example 5/6

- Being completely explicit

```
template <typename Container>
struct my_container {
    using value_t = typename Container::value_type;
    using container_t = Container;
    container_t C;

    my_container(size_t s) : C(s) {}
};

my_container<vector<int, allocator<int>>> my_c(42);
```

An example 6/6

- Customizing the whole

```
template <typename VT,
          template <typename, typename> class CT = vector,
          template <typename> class Allocator = allocator>
struct my_container {
    using value_t = VT;
    using container_t = CT<value_t, Allocator<value_t>>;
    container_t C;

    my_container(size_t s) : C(s) {}
};

my_container<int> my_c(42);
my_container<int, vector> my_c2(42);
my_container<int, vector, allocator> my_c3(42);
```

Extra Example : demangle_helper

- Here's an example of some of the constructs in use
  - Worked example if time permits
- It uses a compiler extension to print types nicely

Link to Compiler Explorer

Concluding remarks

- Whenever you write the same basic code structure multiple times
  - Can you template it?
  - Separation of algorithm/data
- Example: CPU version / GPU version
  - Basically the same, but with some tweaks
  - Abstract most out, and specialize for the special bits
- Traits abstract out small elements of the logic
  - Many small helper `types` that handle typing sub-tasks
  - Zero/Low cost (at least at run-time) abstractions
- Meta-programming allows library developers to create highly tuned code
  - Manipulating types instead of values
  - Fusion of kernels, Unrolling/reordering of loops
  - Specializations for types/data layouts