Agenda: Modern C++

Focussing

- Template Meta-Programming

© 2018: Creative Commons BY-SA

- by: Dipl.-Ing. Martin Weitzel
- for: MicroConsult GmbH Munich

- Example: template_class/demo.cpp
- Example: template_function/demo.cpp
- Example: meta_programming/demo.cpp
- Example: type_calculation/demo.cpp
- Example: value_calculation/demo.cpp
- Example: type_traits/demo.cpp
- Example: function_sfinae/demo.cpp
- Example: class_sfinae/demo.cpp
- Example: parameter_pack/demo.cpp
- Example: variadic_function/demo.cpp
- Example: my_tuple/demo.cpp

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH MicroConsult

Notice to the Reader

- This document supplies the **Guiding Thread** only
    - It is not recommended to be read it "as-is" stand alone
    - The bulk of teaching material is in the live demos
        - created and augmented throughout this course, while
        - **YOU** – the participants - control for each topic
            - the depth of coverage and
            - the time spent on it
- In the electronic version feel free to follow the links
- The Printed version is provided for annotations in hand-writing

---

If you nevertheless want to use this document for self-study, not only to follow the links to the compilable code, **also vary and extend it**.

If you need suggestions what to try (add or modify) you will find some in this presentation and a lot more usually in the example code itself.

There are two basic forms:

- **Conditional code `#if … #else … #endif`**
  - Usually both versions work, demonstrating alternative ways to solve the particular problem at hand
  - Sometimes not all versions are compatible with C++11 but may require C++14 or even C++14 features
- **Commented-out lines – sometimes with alternatives**
  - Sometimes these are shown because they **do not compile** and should indicate that a particular solution that may even seem attractive at first glance is **not** the way to go.
  - Furthermore, if removing the comment often some other line (in close proximity, typically the previous or next one) needs to be commented.

Understanding the code by trying variations is the crucial step to actually internalise the topics covered!

Template Basics

- Class Templates
- Function Templates

Class Templates

- Originally designed for type-generic container classes
  - At Implementation-Time there is a **definition**
  - At Compile-Time **instantiation** follows

http://en.cppreference.com/w/cpp/language/class_template

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Class Template Definition

- At Implementation-Time:
    - Keep some type(s) generic
    - Keep some value(s) generic
- Much like regular class …
    - … starting with template argument list

Example: template_class/demo.cpp

```
class point {
    double xc, yc;
public:
    point(double xc_, double yc_) : xc(xc_), yc(yc_) {}
    double x() const {return xc;}
    double y() const {return yc;}
    double x(double x_) {return xc = x_;}
    double y(double y_) {return yc = y_;}
    point shifted(double xdelta, double ydelta) const;
};
point point::shifted(double xd, double yd) const  {
    return {xc + xd, yc + yd};
}
```

http://coliru.stacked-crooked.com/a/f9b919a12f39f613

- How to parametrize the type of xc, and yc?

Example (cont.): template_class/demo.cpp

```
template<typename T>
class point {
    T xc, yc;
public:
    point(T xc_, T yc_) : xc(xc_), yc(yc_) {}
    T x() const {return xc;}
    // ...
};

template<std::size_t N>
class fstring {
    char fs[N+1];
    void copy(const char* cp) {std::strncpy(fs, cp, N)[N] = '\0';}
public:
    fstring(const char *init) {copy(init);}
    fstring& operator=(const fstring& rhs) {copy(rhs.fs);
                                        return *this;}
    // ...
};
```

- Which of the both classes above templates a type?
- which a compile time constant?
- Could both be combined?
- (For which of both classes would that perhaps make sense?)

Class Template Instantiation

- Concrete template arguments **must** be supplied on instantiation
  - Class name should be thought "including" concrete types, i.e.
  - Same template with different arguments denote different classes

Example (cont.): template_class/demo.cpp

```
template<typename T>
class point {
    T xc, yc;
public:
    point(T xc_, T yc_) : xc(xc_), yc(yc_) {}
    T x() const {return xc;}
    // ...
    T y(T y_) {return yc = y_;}
    point shifted(T xd, T yd) const;
};
template<typename T>
point<T> point<T>::shifted(T xd, T yd) const {
    return {xc + xd, yc + yd};
}

int main() {
    point<double> a{3.5, 7.0};
    point<int> c{10, 20};
    // ...
}
```

- How are arguments transfered – by value or by reference?
- (What if the instantiation type of `point` were *move only*?)

Class Template Specialisation

- Existing *Primary Templates* may be specialised
- Syntactically recognized:
    - Angle brackets follow class name in definition …
        - … holding argument list **identical** to primary template
    - "Outer" argument list must be different ("specialise" something):
        - Empty for a full specialisation
        - Otherwise holds names for deduction
- **Compiler selects "most specialised" matching version**

http://en.cppreference.com/w/cpp/language/template_specialization

http://en.cppreference.com/w/cpp/language/partial_specialization

Example (cont.): template_class/demo.cpp

```cpp
template<typename T> struct tp;

template<> struct tp<int>
{static std::string str() {return "int";}};

template<typename T> struct tp<T*>
{static std::string str() {return tp<T>::str() + "*";}};

template<typename T> struct tp<const T>
{static std::string str() {return "const " + tp<T>::str();}};

int main() {
    PX(tp<int>::str());
    PX(tp<double>::str());
    int i = 42;
    PX(tp<decltype(i)>::str());
    PX(tp<decltype(&i)>::str());
    int cri = i;
    PX(tp<decltype(cri)>::str());
}
```

- Identify the *Primary Template* in the code above.
- Identify its *Full* and *Partial* Specialisations.
- Further flesh-out the code to make it usable for the examples.

Function Templates

- Originally designed for type-generic algorithms

http://en.cppreference.com/w/cpp/language/function_template

Function Template Definition

- At Implementation-Time:
  - Keep some type(s) generic (typical)
  - Keep some value(s) generic (rare but possible)
- Much like regular function …

  … but starts with template argument list

Example: template_function/demo.cpp

First version:

```
template<typename T>
T square(const T& arg) {
    return arg*arg;
}
```

Second version:

```
template<typename T>
auto square(const T& arg) -> decltype(arg*arg) {
    return arg*arg;
}
```

http://coliru.stacked-crooked.com/a/08d2a5bf1ceffd78

- Describe what is different between both versions of `square`.
- Under which circumstances may this become visible to the caller?
- Is it wise handing over arguments per reference for **all** types?
- (How might the latter be changed for **some** types?)

Function Template Instantiation

- Concrete argument values **may** be supplied …
- … but more often are deduced from call arguments
    - Generic name in argument list may be adorned
    - **Be sure to understand what exactly is deduced**[*]

```
template<typename T>
void foo(T arg) {
    … // type of `T` same
    … // as type of `arg`
}
template<typename T>
void bar(const T& arg) {
    … // what is the type
    … // of `arg` here ...?
    … // NOT same as `T`!
}
```

Assume calls like:

```
int i = 10;
foo(i); bar(i);

const int k = 20;
foo(k); bar(k);

foo(std::atof("10.1"));
bar(std::atof("10.2"));
```

*: Type deduction for template arguents is not limited to trvial or close to trial cases as is shown in the examples given here: http://coliru.stacked-crooked.com/a/96c78740a1c689f5

Example (cont.): template_function/demo.cpp

```
int main() {
    PX(square(2));      PX(square<double>(2));
    PX(square(2.5));    PX(square<int>(2.5));

//  PX(square(false));  PT(decltype(square(false)));
//  PX(square(true));   PT(decltype(square(true)));

//  PX(square<double>(true));

}
```

- Predict the output of the above code.
- Will the currently commented-out lines compile too?
- Or not all but some?
- (If so, what output do you expect?)

Function Template Overloading

- Function templates may be overloaded by non-templates
    - **Exactly matching overloads get always preferred**
- In addition there may also be templated overloads
    - If any, the "closest match" gets selected …
    - **… or Compile-Error if not unique**
- Finally, function templates may also be **fully** specialised

http://en.cppreference.com/w/cpp/language/overload_resolution

http://en.cppreference.com/w/cpp/language/template_specialization

---

Note: there is no syntax for partial specialisation of function templates – maybe because the rules are complicated enough even without … :-/

http://www.gotw.ca/gotw/049.htm

Example (cont.): template_function/demo.cpp

```
template<typename T>
auto square(const T& arg) -> decltype(arg*arg) {
    return arg*arg;
}

auto square(bool arg) -> bool {
    return arg;
}

int main() {

    PX(square(false));  PT(decltype(square(false)));
    PX(square(true));   PT(decltype(square(true)));

    PX(square<double>(true));
}
```

- Why is it that both versions of `square` can coexist?
- Will the existence of the second version change the output?

Meta-Programming

- The key insight is:
    - any instantiation of a template requires that
    - the compiler carries out some internal calculation

Example: meta_programming/demo.cpp

```
// template definitions for `add_ptr`, `remove_ptr`,
// and `remove_all_ptr` shown and explained later

int main() {
//                                     equivalent to:
    … my::add_ptr<int>::type …
//    ^^^^^^^^^^^^^^^^^^^^^^------------ type `int*`

    … my::remove_ptr<int**>::type
//    ^^^^^^^^^^^^^^^^^^^^^^^^^^^------- type `int*`

    … my::remove_all_ptr<int***>::type …
//    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^-- type `int`

    … my::countbits<42>::value …
//    ^^^^^^^^^^^^^^^^^^^^^^^^---------- value `42`
}
```

http://coliru.stacked-crooked.com/a/e926a38155979659

- Where's the "meta" – can you see it?
- Understand that all "calculations" happen at *Compile Time*.
- Why is appending the `::type` or `::value` part absolutely essential?

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Meta-Programming Basics

- The C++ template system is a full programming language …
    - … yet often not the most convenient one
    - (at least given for what it was used in the past)
- But inconvenience is at the site of the implementor …
    - … to ease the work of its clients
    - **except for messy error messages**

Be sure to understand: anything talked about with respect to Meta-Programming means things that happen at Compile-Time, though often an executable program needs to be created too, but simply to show the effect.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): meta_programming/demo.cpp

```
template<typename T>
struct add_ptr {using type = T*;};

template<typename T> struct remove_ptr;
template<typename T>
struct remove_ptr<T*> {using type = T;};

template<typename T> struct remove_all_ptr {using type = T;};
template<typename T> struct remove_all_ptr<T*> {
    using type = typename remove_all_ptr<T>::type;
};

template<unsigned N>
struct countbits {
    static const std::size_t value = (N & 0x1)
                                   + countbits<(N>>1)>::value;
};
template<>
struct countbits<unsigned{0}> {
    static const std::size_t value = 0;
};
```

- Identify the *Meta Functions* above and their arguments.
- How are they are getting "called" and what do they return?

Types as Meta-Programming Input and Output

- Any "input" to a meta program is
  - either a type
  - or a value
- Needs to explicitly stated somewhere in the source code

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): meta_programming/demo.cpp

```
int main() {
//  ----------------------------------------------- "input" is:

//  typeprinter< add_ptr<int>::type           > notused;
//                       ^^^------------------------------ ??

//  typeprinter< remove_ptr<int**>::type      > notused;
//                          ^^^^^------------------------- ??

//  typeprinter< remove_all_ptr<int***>::type > notused;
//                              ^^^^^^-------------------- ??

    str::cout << countbits<42>::value;
//                         ^^----------------------------- ??
}
```

- Locate the "input" of each *Meta Function* and determine what it is.
  - A type?
  - Or a value?
- Assuming the *Meta Functions* do what their names suggests:
  - What is the expected "output"?
  - What is the difficulty with showing a type?

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Template Meta-Programming Output

- Any "output" of a meta program is
  - "something that can be compiled"
  - or an error message

For demonstration purposes meta programming often includes a small program that can be compiled to an executable, showing the intended effect (e.g. by printing a calculated type).

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): meta_programming/demo.cpp

```
template<typename> struct typeprinter;

int main() {
//  ----------------------------------------------- "output" is:

//  typeprinter< add_ptr<int>::type           > notused;
//               ^^^^^^^^^^^^----------------------------- ??

//  typeprinter< remove_ptr<int**>::type      > notused;
//               ^^^^^^^^^^^^^^^^^------------------------ ??

//  typeprinter< remove_all_ptr<int***>::type > notused;
//               ^^^^^^^^^^^^^^^^^^^^^^------------------- ??

    str::cout << countbits<42>::value;
//               ^^^^^^^^^^^^^---------------------------- ??
}
```

- How is *Meta Program* "value output" shown in the example above?
- How is *Meta Program* "type output" shown in the example above?

Digression: Recipe to Implement a Type Printer

- Instead of just defining a variable (`notused`):
  - define a full specialisation for `int` …
  - … (and any other basic type that may occur) …
  - … with a static member returning the type as string value
  - add partial specialisations for adorned types …
  - … delegating further resolution to recursive calls

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH MicroConsult

Optional Example (cont.): meta_programming/demo.cpp

```
template<typename> struct typeprinter;

template<> struct typeprinter<int>
{static std::string str() {return "int";}};
// ... and more "mutatis mutandis"
template<typename T> struct typeprinter<T*>
{static std::string str() {return typeprinter<T>::str() + "*";}};
// ... and more "mutatis mutandis"

int main() {
    std::cout
    << typeprinter<add_ptr<int>::type>::str()           << '\n'
    << typeprinter<remove_ptr<int**>::type>::str()      << '\n'
    << typeprinter<remove_all_ptr<int***>::type>::str() << '\n'
    <<             countbits<42>::value                 << '\n';
}
```

- For practical purposes consider to wrap output of a
    - … type into a macro accepting a type as argument,
    - … value into a macro accepting an expression as argument.

The macros PT and PX that have been used a lot already do just that.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Optional Example (cont.): meta_programming/demo.cpp

```
#define PX(expr)\
    ((void)(std::cout << __FUNCTION__  << ':' << __LINE__\
                      << "\t" #expr " --> "\
                      << (expr) << std::endl))
#define PT(type)\
    ((void)(std::cout << __FUNCTION__  << ':' << __LINE__\
                      << "\t" #type " --> "\
                      << typeprinter<type>::str() << std::endl))
int main() {
    PT(add_ptr<int>::type);
    PT(remove_ptr<int**>::type);
    PT(remove_all_ptr<int***>::type);
    PX(countbits<42>::value);
}
```

- Over what was describe so far:
  - Which refinement is contained in the macros?
  - Why does it makes sense?
  - Can you still spot any shortcomings?

Implementation Selection

- Typical meta programming goal is "implementation selection"
    - In simple cases with (direct) specialisation
    - Alternatively with overloading and SFINAE
- E.g. choose between
    - value and reference argument in a function call
    - member-wise copy and `std::memcpy` in a container

This page mainly aims to the inpatient who at least want to have a slight idea where all this "meta ..." leads to.

Template Meta-Programming Functions

- Any template class also constitutes a (meta-) function
  - its arguments come from the template argument list
  - it is called by instantiating the template
- Calling conventions make sense, e.g. a member
  - ...`::result` that "calls" the meta function, returning
    - *either* a type
    - *or* a value
  - ...`::type` that "calls" the meta function, returning a type
  - ...`::value` that "calls" the meta function, retuning a value

Sticking to *one* of the above – i.e. *either* using `::result` for both, types and values returned, *or* differentiating with `::type`` and ``::value` – makes a lot of sense, because it enables "higher order" meta functions (that are meta-functions which manipulate other meta functions).

Branches in Meta-Programs

- **All** branching in Meta-Programming
    - needs to be done by specialisation
- I.e. there needs to be a primary template …
    - … **either** covering the general case …
    - … **or** just being declared but not defined …
    - … if there are specialisations for all possible cases

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): meta_programming/demo.cpp

```
namespace my {
    template<typename T>
    struct is_ptr {static const bool value{false};};
    template<typename T>
    struct is_ptr<T*> {static const bool value{true};};

    template<typename T>
    struct remove_ptr;
    template<typename T>
    struct remove_ptr<T*> {using type = T;};
}

int main() {
                 PX(my::is_ptr<int>::value);
             //  PT(my::remove_ptr<int>::type);
                 PX(my::is_ptr<int*>::value);
                 PT(my::remove_ptr<int*>::type);
    int i{42};   PT(my::is_ptr<decltype(i)>::value);
                 PT(my::remove_ptr<decltype(&i)>::type);
}
```

- Explain branches `my::is_ptr` and `my::remove_ptr`.
- Why does `my::rm_pointer` not work with non-pointers?
- (Feel free to add more meta function test "calls" to the code.)

Template Meta-Programming Loops

- In C++98 any meta programming loops
  - need to resolved to recursion
  - terminated by specialisation

Only for parameter packs (variadic templates) this is slightly released in C++11 and further generalised to fold expressions in C++1z.

Example (cont.): meta_programming/demo.cpp

```
namespace my {
    template<typename T>
    struct remove_all_ptr {using type = T;};
    template<typename T>
    struct remove_all_ptr<T*> {
    //  using type = remove_all_ptr<T>;
        using type = typename remove_all_ptr<T>::type;
    };
}

int main() {
                 PT(my::remove_all_ptr<int>::type);
                 PT(my::remove_all_ptr<int*>::type);
                 PT(my::remove_all_ptr<int****>::type);
    int *p;      PT(my::is_ptr<decltype(&p)>::value);
                 PT(my::remove_all_ptr<decltype(&p)>::type);
}
```

- Explain the loop in `my::remove_all_ptr`.
- What is essential to avoid an infinite loop?
- How does `my::remove_all_ptr` handle non-pointers?
- How does this differ from handling non-pointers in `my::remove_ptr`?
- How to make one pointer level mandatory in `my::remove_all_ptr` too?
- (Why compiles the commented out flawlessly but still does not work?)

Compile-Time Type Calculations

- Internal calculations may transform types
    - There are many practical uses …
    - … though mostly not visible at first glance

Please keep calm, be patient, and follow the examples, even if you have no idea to where it finally leads.

Example: type_calculation/demo.cpp

```
template<typename T>
struct fixpoint {
    const T x{};
    const T y{};
//  using argtype = T;          // for register-types
    using argtype = const T&;  // for non-register types
    fixpoint(argtype x_, argtype y_)
        : x(x_), y(y_) {
        PT(argtype);
    }
    fixpoint shifted(argtype xd, argtype yd) {
        PT(argtype);
        return {x + xd, y + yd};
    }
};
```

http://coliru.stacked-crooked.com/a/8e1d4212337eb36b

- How could `argtype` be defined so that it `automatically` selects
    - T for types fitting into a register and
    - `const T&` for all other types?

You may arbitrarily assume here which types fit into a register, e.g. `short`, `int`, and `float` do, while `long`, `long long`, `double`, and `long double` don't.

Practical Uses of Type Calculations

- Remember: All type calculations
  - finally result in implementation selection
  - are an and in itself only in simple demo programs

Example (cont.): type_calculation/demo.cpp

```cpp
template<typename T>
struct argument_type {using type = const T&;};
template<> struct argument_type<short> {using type = short;};
template<> struct argument_type<int> {using type = int;};
template<> struct argument_type<float> {using type = float;};

template<typename T>
struct fixpoint {
    const T x{};
    const T y{};
    using argtype = typename argument_type<T>::type;
    fixpoint(argtype x_, argtype y_)
        : x(x_), y(y_) {
    }
    fixpoint shifted(argtype xd, argtype yd) {
        PT(argtype);
        return {x + xd, y + yd};
    }
};
```

- Explain how the meta function `argument_type` optimises the code.
- Add code to demonstrate the above **actually works**.
- Why aren't there any specialisations or `double` etc. necessary?
- (And what happens with `char` … or all the `unsigned` … types?)

Standard Type Transformations

- Frequently required transformation are provided
    - as part of the standard type traits
    - especially for *Iterators* via `std::iterator_traits`

---

http://en.cppreference.com/w/cpp/header/type_traits
(From section *Const-volatility specifiers* to section *Miscellaneous transformations*)

http://en.cppreference.com/w/cpp/iterator/iterator_traits

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): type_calculation/demo.cpp

```
template<typename InIt, typename OutIt>
OutIt noduplicates(InIt from, InIt upto, OutIt dest) {
    using element_type =
          typename std::iterator_traits<InIt>::value_type;
    std::set<element_type> seen;
    while (from != upto) {
        if (seen.insert(*from).second)
        *dest++ = *from;
        ++from;
    }
    return dest;
}

int main() {
    std::vector<int> test_data = {1, 4, 2, 44,  4, 4, 3, 2, 3};
    noduplicates(std::begin(test_data), std::end(test_data),
                 std::ostream_iterator(std::cout, " "));
    std::cout << std::endl;
}
```

- Does the code also work with `element_type = InIt::value_type`?
- With `element_type = typename InIt::value_type`?
- Even if `test_data` is a native array instead of an `std::vector`?
- (Unrelated: How does noduplicates differ from `std::unique_copy`?)

Compile-Time Value Calculations

- For value calculations in meta programming the choices are:
  - `constexpr` functions (since C++11)
  - Templates (meta functions) with a `static const` member
- To unify both also values may be turned into types

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: value_calculation/demo.cpp

```cpp
constexpr unsigned gcd_function(unsigned m, unsigned n) {
    return n ? gcd_function(n, m % n) : m;
}

int main() {
    PX(gcd_function(42, 8));
    PX(gcd_function(42, 7));
    PX(gcd_function(42, 6));
    PX(gcd_function(42, 5));
    PX(gcd_function(42, 4));
    PX(gcd_function(42, 2));
    PX(gcd_function(6, 7));
    PX(gcd_function(7, 6));
    PX(gcd_function(126, 7));
    PX(gcd_function(126, 6));
    PX(gcd_function(126, 1));
}
```

http://coliru.stacked-crooked.com/a/b8fb5d149600d9bb

- What is the restriction to the call arguments of `gcd_function`?
- What happens if that restriction is not met?

Template Meta-Programming vs. `constexpr` Functions

- Replacing a `constexpr` function with a template requires …
  - … to turn all loops into recursion …
  - … and branches into specialisation.
- Without training this may often cause headaches to novices.
- Some experience with *Functional Programming* will **surely** help

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): value_calculation/demo.cpp

```cpp
template<unsigned M, unsigned N>
struct gcd_template {
    static constexpr unsigned value =
                     gcd_template<N, M % N>::value;
};
template<unsigned M>
struct gcd_template<M, 0> {
    static constexpr unsigned value = M;
};

int main() {
    PX(gcd_template<42, 8>::value);
    // ...
    PX(gcd_template<42, 8>);
//  PT(gcd_template<42, 8>);
//  PT(gcd_template<42, 8>::type);
}
```

- Test the above with a main program that "calls" `gcd_template`.
- What if the arguments are no compile time constants?
- What if you omit `::value`?
- Replace with `::type`?
- Change PX to PT?

Dealing with Values as Types

- There is a way to unify constant values with types:
  - Wrap it into a template!
  - Either a generic one generalising type, like
    - `template<typename T, T I> struct constant …`
  - Or type specific once wrapping the value only, like
    - `template<int I> struct integer …`
    - `template<bool B> struct boolean …`
- Floating types are **not** usable as template parameters

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): value_calculation/demo.cpp

```
namespace my {
    template<bool B>
    struct boolean {
    //  using type = bool;
        static constexpr bool value = B;
    };

    template<typename T1, typename T2>
    struct is_same : boolean<false> {};
    template<typename T1>
    struct is_same<T1, T1> : boolean<true> {};
}

int main() {
    PX(my::is_same<double, char>::value);
    PX(my::is_same<char, char>::value);
    PX(my::is_same<char, signed char>::value);
    // ...
    PT(my::is_same<double, char>::type);
    // ...
}
```

- Why could it make sense to add a nested definition `type`?
- And `value_type`?

Type Traits

- In meta programming with the term *Traits* is used for …
    - … attaching some (meta) information to a type
    - more general they are the equivalent to *Data Bases*
- Traits usually are stored in header files …
- … and made available by `#include`-ing these

Example: type_traits/demo.cpp

```
// assume there is the need to attach information to basic types:
// - what is the raw memory equivalent for its storage
// - whether it fits into a register

int main() {
    // --------------------------------- first form
    PX(fits_register<short>::value);
    PX(fits_register<int>::value);
    PX(fits_register<double>::value);
    // ...
    PT(raw_equivalent<double>::type);
    // --------------------------------- second form
    PX(bt_info<short>::fits_reg);
    // ...
    PT(bt_info<int>::raw_equiv);
    PT(bt_info<float>::raw_equiv);
    PT(bt_info<double>::raw_equiv);
}
```

http://coliru.stacked-crooked.com/a/27f12f6e0e51fe47

- Understand the above code shows two **alternatives** (for comparison).
- (Which form do you like better?)

Storing Compile-Time Data

- Think of a database organized
  - in columns (kind of data)
  - and rows (accessed by key)
- Row is selected by a type
- Kind of data is detail (information) about this type

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH MICROCONSULT

Access by (`struct`) Name

- There may be **one** meta function for **each** columns
    - Type (to be used as "key") goes into the argument list
    - Kind of information is determined by the (function) name
- (somewhat like specific functions in Run-Time programming)
- Examples from standard type traits:
    - (too many to list)

http://en.cppreference.com/w/cpp/header/type_traits

Example (cont.): type_traits/demo.cpp

```
// creating the "data base" (first form)

template<typename T> struct fits_reg       : boolean<false> {};
template<> struct fits_reg<short>          : boolean<true> {};
template<> struct fits_reg<unsigned short> : boolean<true> {};
template<> struct fits_reg<int>            : boolean<true> {};
template<> struct fits_reg<unsigned int>   : boolean<true> {};

template<typename T> struct raw_equiv;
template<> struct raw_equiv<short>  {using type = unsigned int;};
template<> struct raw_equiv<int>    {using type = unsigned int;};
template<> struct raw_equiv<long>   {using type = unsigned long;};
template<> struct raw_equiv<float>  {using type = unsigned int;};
template<> struct raw_equiv<double> {using type = unsigned long;};
```

- What is the default for `fits_reg`?
- What is the default for `raw_equiv`?
- How to add another trait, say `has_raw_equiv`?

Selection by (`struct`) Member

- There may be **one** meta function for **all** columns
    - Type (to be used as "key") goes into the argument list
    - Kind of information determined by "calling convention"
- (somewhat like member functions in Run-Time programming)
- Examples from standard type traits:
    - `std::iterator_traits`
    - `std::numeric_limits`
    - (and many more)

http://en.cppreference.com/w/cpp/iterator/iterator_traits#Member_types

http://en.cppreference.com/w/cpp/types/numeric_limits#Member_constants

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH MICROCONSULT

Example (cont.): type_traits/demo.cpp

```
// creating the "data base" (second form)

template<typename T> struct bt_info;

template<> struct bt_info<short> {
    static const bool fits_reg{true};
    using raw_equiv = unsigned int;
};
template<> struct bt_info<unsigned short> {
    static const bool fits_reg{true};
    using raw_equiv = unsigned int;
};
// ...
template<> struct bt_info<double> {
    static const bool fits_reg{false};
    using raw_equiv = unsigned long;
};
```

- Could defaults here be introduced too? (Somehow?)
- Does `const` or `constexpr` make more sense for `fits_reg`?
- How to add another trait, say has_raw_equiv?

(More) Standard Traits

- C++11 defines many more standard type traits
  - Purpose of many can be derived from their name
  - Nevertheless lookup docs to avoid misinterpretation
- When used indirectly (in a template)
  - for accessing the …`::type`-member …
  - … be sure to prepend `typename`

http://en.cppreference.com/w/cpp/header/type_traits

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH MicroConsult

Example (cont.): type_traits/demo.cpp

```
int main() {
    int i{0};              PT(decltype(i));
                           PT(decltype(*&i));
    const auto &cri = i;  PT(decltype(cri));
                           PT(decltype(*&cri));

    PX(std::is_const<decltype(cri)>::value);
    PT(std::remove_const<decltype(cri)>::type);
    PX(std::is_reference<decltype(cri)>::value);
    PT(std::remove_reference<decltype(cri)>::type);

    PX(std::is_const<decltype(i)>::value);
    PX(std::is_reference<decltype(i)>::value);

    PX(std::is_same<signed int, int>::value);
    PX(std::is_same<signed char, char>::value);
    PX(std::is_same<unsigned char, char>::value);
}
```

- Run the above code and explain its output.
- Can `const` be removed from a non-`const`?
- Or a reference from a non-reference?
- What about the last three lines?

Abbreviating Type Access in C++14

- In addition to the ...`::type` calling convention ...
    - ... C++14 adds type aliases ending in `_t`
    - (reduces noise by avoiding `typename` ...`::type`)

http://en.cppreference.com/w/cpp/header/type_traits
(see `using` type aliases ending with `_t`)

Example (cont.): type_traits/demo.cpp

```
int main() {
    int i{0};             PT(decltype(i));
                          PT(decltype(*&i));
    const auto &cri = i;  PT(decltype(cri));
                          PT(decltype(*&cri));
// --------------------------------------------- since C++11
    PT(std::remove_const<decltype(i)>::type);
    PT(std::remove_const<decltype(cri)>::type);
    PT(std::remove_reference<decltype(i)>::type);
    PT(std::remove_reference<decltype(cri)>::type);
    PT(std::conditional<true, int**, double&>);
// --------------------------------------------- since C++14
    PT(std::remove_const_t<decltype(i)>);
    PT(std::remove_const_t<decltype(cri)>);
    PT(std::remove_reference_t<decltype(i)>);
    PT(std::remove_reference_t<decltype(cri)>);
    PT(std::conditional_t<true, int**, double&>);
}
```

- Be sure to understand the relation between the usage forms.

Abbreviating Value Access in C++14

- In addition to the …`::value` calling convention …
  - … C++14 adds
    - `constexpr` constructor and `operator bool()`
    - (reduces noise by avoiding access to …`::value`)
  - … C++1z adds
    - template variable …_v
    - (reduces noise by avoiding access to …`::value`)

http://en.cppreference.com/w/cpp/header/type_traits
(all `is_`… traits)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): type_traits/demo.cpp

```
int main() {
    int i{0};         PT(decltype(i));
                          PT(decltype(*&i));
    const auto &cri = i;  PT(decltype(cri));
                          PT(decltype(*&cri));
// --------------------------------------------- since C++11
    PX(std::is_const<decltype(i)>::value);
    // ...
    PX(std::is_reference<decltype(cri)>::value);
    PX(std::is_same<signed int, int>::value);
// --------------------------------------------- since C++14
    PX(std::is_const<decltype(i)>{});
    // ...
    PX(std::is_reference<decltype(cri)>{});
    PX(std::is_same<signed int, int>{});
// --------------------------------------------- since C++17
    PX(std::is_const_v<decltype(i)>);
    // ...
    PX(std::is_reference_v<decltype(cri)>);
    PX(std::is_same_v<signed int, int>);
}
```

- Be sure to understand the relation between the usage forms.

The SFINAE Principle

- SFINAE = Substitution Failure Is Not an Error
    - Selects overload on finer-grained criteria
- Strategy
    - Specify an ambiguous overload set
    - Make all instantiations **except one** fail
- Common idiom for testing existence of operations
    - Comma separated expression in `decltype` as function result

http://en.cppreference.com/w/cpp/language/sfinae

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH MicroConsult

`std::enable_if`

- May be considered a "more readable" alternative
    - First argument is condition (Compile-Time evaluated `bool`)
    - Second argument is a type (defaults to `void`)
- Typical use:
    - C++11: `… typename std::enable_if< …cond… , …type… >::type …`
    - C++14: `… std::enable_if_t< …cond… , …type… >`
- Expands to
    - …type… if …cond… is true
    - substitution failure otherwise (but: SFINAE!)

http://en.cppreference.com/w/cpp/types/enable_if

SFINAE for Functions

- Applying SFINAE to functions can be done via
  - … function return type
  - … extra argument
  - … template value parameter
  - … template type parameter

Example: function_sfinae/demo.cpp

```
template<typename T>
void foo_integral_unsigned(T arg) { /* ... */ }

template<typename T>
void foo_integral_signed(T arg) { /* ... */ }

void foo(signed short arg)   {foo_integral_signed(arg);}
void foo(unsigned short arg) {foo_integral_unsigned(arg);}
void foo(signed int arg)     {foo_integral_signed(arg);}
// ...
void foo(signed long long arg)    {foo_integral_signed(arg);}
void foo(unsigned long long arg)  {foo_integral_unsigned(arg);}

int main() {
    foo(42u); // --> foo_ integral_unsigned with T = unsigned int
    foo(42LL); // --> foo_integral_signed with T = long long
}
```

Coliru

- Understand that the templates (`foo_…`) themselves are "too broad" …
- … and how the problem is solved by overloaded delegations.
- How many delegations might need to be written in practice.
- What if there is a third template for all floating point types?

SFINAE via Return Type

- Use `enable_if` as function return type
    - either in classic syntax (left) to function name
    - or in trailing return type syntax
- Probably the "easiest" and mostly used style
    - but impossible for constructor and destructor

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH MICROCONSULT

Example (cont.): function_sfinae/demo.cpp

```
template<typename T>
typename std::enable_if<
   std::is_unsigned<T>::value
>::type foo(T arg) {
    /* ... */
}

template<typename T>
typename std::enable_if<
   std::is_signed<T>::value && !std::is_floating_point<T>::value
>::type foo(T arg) {
    /* ... */
}

template<typename T>
typename std::enable_if<
   std::is_floating_point<T>::value
>::type foo(T arg) {
    /* ... */
}
```

- What if there is a return value, i.e. something else but void?
- Why is a test `!std::is_floating_point` added in the second case?
- How about readability of the above - any ideas to improve it?

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH MicroConsult

SFINAE via Extra Argument

- Add an extra argument (not accessed in the function)
  - Define its type with `enable_if`
  - Supply default value
- Still "relatively" straight forward
  - **Only** option for constructors
  - **No* option for overloaded operators

Example (cont.): function_sfinae/demo.cpp

```
template<typename T>
void bar(T arg,
         typename std::enable_if<
             std::is_signed<T>::value
         >::type* = nullptr
) {
    /* ... */
}

template<typename T>
void bar(T arg,
         typename std::enable_if<
         //  std::is_unsigned<T>::value
             !std::is_signed<T>::value
         >::type* = nullptr
) {
    /* ... */
}
```

- Identify the *sfinae*-ed argument.
- What is its type (if there is no substitution failure).
- (Why turn it into a pointer?)

SFINAE via Template argument

- Via template type argument:
  - Add unused last template argument
  - Set default with `enable_if` to type `void`
- Via template value argument:
  - Define as `void *` (using `enable_if` for `void`)
  - Set default with `enable_if` to `nullptr`
- More "arcane" and probably harder to understand

SFINAE for Classes

- SFINAE may be used to specialise class templates
- Strategy:
  - Add trailing (otherwise unused) `typename` argument
  - Initialize with default (e.g. void)
  - Specialisation make initialisation succeed or fail

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: class_sfinae/demo.cpp

```
// primary template
template<typename, typename = void>
class number;

template<typename T>
class number<T, std::enable_if_t<std::is_integral<T>{}>> {
public:
    static auto str() {return "integral type";}
};

template<typename T>
class number<T, std::enable_if_t<std::is_floating_point<T>{}>> {
public:
    static auto str() {return "floating type";}
};

int main() {
    PX(number<int>::str());     PX(number<unsigned long>::str());
    PX(number<float>::str());   PX(number<long ouble>::str());
}
```

http://coliru.stacked-crooked.com/a/b739113d09f09635

- Try more types – including bool and char.

Variadic Templates

Defining Parameter Packs

- Parameter packs are defined by adding an ellipsis (three dots)
  - e.g.: `template<typename... Ts>` …
  - or: `template<int... Vs>` …
- Value lists may refer to a typename mentioned earlier, e.g.
  - e.g. `template<typename T, T... Vs>` …
- **Pack must come last in the template argument list**

http://en.cppreference.com/w/cpp/language/parameter_pack

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: parameter_pack/demo.cpp

– will be given live –

http://coliru.stacked-crooked.com/a/00d2528de2ac01ec

Number of Elements in Parameter Pack

- `sizeof...` is a Compile-Time function returning the pack size
    - e.g. `sizeof...(Ts)`
    - or `sizeof...(Vs)`

Do not confuse `sizeof...` (which returns the element count) with classic `sizeof` (which returns the number of bytes in memory).

Emulating Parameter Packs

- Possible to some degree
- Requires much systematic code
    - Boresome and error prone to write
    - Difficult to understand
    - Hard to maintain
    - Inefficient to compile

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH MicroConsult

Compile-Time Only Data

- Parameter packs can represent "pure" Compile-Time data
    - e.g. a list of integral values
        - only existing in the compiler memory
        - not in the programs executable

Note: There are practical applications but the following example should rather demonstrate the idea only:

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): parameter_pack/demo.cpp

– will be given live –

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Compile-Time Type Lists

- Often parameter packs represent lists of types
    - Such lists may only exist at Compile-Time …
    - … but also may be used to define Run-Time data

A practical application may be to select the best matching type from a list of given types.

Example (cont.): parameter_pack/demo.cpp

– will be demonstrated live –

Boost MPL

- Meta-Programming Library, supporting
  - STL-like Compile-Time data structures
  - STL-like Compile-Time algorithms

http://www.boost.org/doc/libs/release/libs/mpl/doc/index.html

Example (cont.): parameter_pack/demo.cpp

– will be demonstrated live –

Adding Run-Time Data to Type List

- Run-Time Data may be based on Type Lists:
    - Mainly intended for type-safe variadic functions
    - Variadic Class (data-) members also possible
        - e.g. `std::tuple`
        - or `boost::variant`
- Key is "unpacking" a parameter pack with `...`
    - usually combined with Compile-Time recursion

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Type-Safe Variadic Functions

- Classic (C-Style) variadic functions **are not type-safe!**
    - With templates and parameter packs this can be fixed
    - Loops must be resolved into compile time recursion …
    - … which must be terminated by specialisation

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: variadic_function/demo.cpp

```cpp
// NOTE: variable argument list must be terminated with `0uLL`
//
unsigned long long sum_ints(unsigned long long first, ...) {
    using namespace std;
    unsigned long long result = first;
    va_list ap;
    va_start(ap, first);
    while (auto v = va_arg(ap, unsigned long long))
        result += v;
    return result;
}

int main() {
    PX(sum_ints(6uLL, 7uLL, (unsigned long long)(6 * 7), 0uLL));
//  PX(sum_ints(6uLL, 7.0, (6 * 7), 0uLL));
//  PX(sum_ints(6, 7, (6 * 7), 0));
//  PX(sum_ints(6, 7, (6 * 7)));
}
```

http://coliru.stacked-crooked.com/a/28201c30df7e03d1

- What makes `sum_ints` a classic *C-Style Variadic Function*?
- Can you spot potential problems in the commented-out calls?
- Can you spot any problems inside the function `sum_all` itself?

Example (cont.): variadic_function/demo.cpp

– will be demonstrated live –

http://coliru.stacked-crooked.com/a/2a8a0d561b6804aa

Member Data According to Type List

- To add class member data according to a type list …
  - … use a recursive data structure
  - or write many (many, many ...) specialisations

Writing some (few) specialisations may serve as good starting point to recognise a recursively repeating structure (data or code).

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: my_tuple/demo.cpp

```
namespace my {
    template<typename... Ts> struct tuple;

    template<>
    struct tuple<> {};
}

int main() {
    my::tuple<> t0;
    my::tuple<int> t1;
}
```

http://coliru.stacked-crooked.com/a/751d1eea4423a98e

- Identify the primary template in the example above
- How many specialisations exist?
- Which of so usages will compile if there were no other?

The class `tuple` above is defined in namespace `my::` to avoid accidental collisions with `std::tuple`. For space efficiency the enclosing namespace block will not be shown any more when the example continues.

Example (cont.): my_tuple/demo.cpp

```
template<typename T1>
struct tuple<T1> {
    T1 m1;
    tuple(const T1& m1_) : m1(m1_) {}
};
template<typename T1, typename T2>
struct tuple<T1, T2> {
    T1 m1;
    T2 m2;
    tuple(const T1& m1_, const T2& m2_) : m1(m1_), m2(m2_) {}
};
template<typename T1, typename T2, typename T3>
struct tuple<T1, T2, T3> {
    T1 m1;
    T2 m2;
    T3 m3;
    // ...
};
```

- Recognise the systematic approach adding data members.
- How would the constructor for the tuple with three members look?
- (Unrelated: What could be improve with respect to its arguments?)
- **Why does this approach not fit well to *Meta Programming*?**

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH MicroConsult

Unpacking Parameter Packs

- Unpacking a parameter pack is requested by an ellipsis (three dots)
  - The ellipsis occur to the right of an expression
  - The expression needs to contain at least one parameter pack
  - It is repeated by substituting elements from the pack
- Valid unpacking contexts are:
  - Function call parameter – **no** left to right guarantee
  - Aggregate initialisation – **left to right guarantee**
- More that one parameter pack can be unpacked simultaneously

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH MicroConsult

Example (cont.): my_tuple/demo.cpp

```
template<typename... Ts> struct tuple;

template<>
struct tuple<> {};

template<typename T1, typename... Ts>
struct tuple<T1, Ts...> {
    T1 m1;
    tuple<Ts...> ms;
    tuple(const T1& m1_, const Ts&... ms_) : m1(m1_), ms(ms_...) {}
};
```

- Where is `...` used above to **define a name** for parameter pack?
- Where is it applied to **unpack** a parameter pack?
- Where goes the first and where goes all the rest (if any)?
- How do the members get initialised? (And how accessed?)

The above works well but is not space efficient, as – for technical reasons – the empty tuple (`tuple<>`) still needs at least one byte in data memory. In practice the remaining elements are therefore rather put into a base class instead of a member. (The conversion from one approach to the other is quite systematic and left as an exercise to the reader.)

Example (cont.): my_tuple/demo.cpp

```
template<std::size_t I, typename T>
auto get(const T& arg)
    -> typename std::enable_if<I != 0, ........>::type {
                                       ^^^^^^^^ -------- ??
    return get<I-1>(arg.ms);
}
template<std::size_t I, typename T>
auto get(const T& arg)
    -> typename std::enable_if<I == 0, .........>>::type {
                                       ^^^^^^^^^ ------- ??
    return arg.m1;
}
```

- Explain the recursive approach, ignoring the ...... (for a moment).
- **Now: what has to go in the still omitted parts?**

C++11 has a meta function for `std::tuple` to do what is required:

http://en.cppreference.com/w/cpp/utility/tuple/tuple_element

In this example it will be built as next step.

Example (cont.): my_tuple/demo.cpp

```
template<std::size_t I, typename T> struct ith_type;
// ------------------------------------------------------ 1st
template<typename T1>
struct ith_type<0, tuple<T1>> {using type = T1;};

template<typename T1, typename T2>
struct ith_type<0, tuple<T1, T2>> {using type = T1;};

template<typename T1, typename T2, typename T3>
struct ith_type<0, tuple<T1, T2, T3>> {using type = T1;};
// ------------------------------------------------------ 2nd
template<typename T1, typename T2>
struct ith_type<1, tuple<T1, T2>> {using type = T2;};

template<typename T1, typename T2, typename T3>
struct ith_type<1, tuple<T1, T2, T3>> {using type = T2;};
// ------------------------------------------------------ 3rd
template<typename T1, typename T2, typename T3>
struct ith_type<2, tuple<T1, T2, T3>> {using type = T3;};
```

- Do you recognise a pattern? (By help of the separator lines?)
- Which amount of work will be required to scale-up this approach?
- **Now again:** Where to look for the rescue with *Meta Programming*?

Example (cont.): my_tuple/demo.cpp

```
template<std::size_t I, typename T> struct ith_type;
//  ------------------------------------------------------------
template<typename T1>
struct ith_type<0, tuple<T1>> {using type = T1;};
//  ------------------------------------------------------------
template<typename T1, typename... Ts>
struct ith_type<0, tuple<T1, Ts...>> {
    using type = T1;
};
//  ------------------------------------------------------------
template<typename T1, typename T2, typename... Ts>
struct ith_type<1, tuple<T1, T2, Ts...>> {
    using type = T2;
};
//  ------------------------------------------------------------
template<typename T1, typename T2, typename T3, typename... Ts>
struct ith_type<2, tuple<T1, T2, T3, Ts...>> {
    using type = T3;
};
```

- What is the advantage of this solution?
- Why doesn't it still scale perfectly?
- What is the answer – **who is your friend**?

Example (cont.): my_tuple/demo.cpp

```
template<std::size_t I, typename T>
struct ith_type;

template<std::size_t I, typename T1, typename... Ts>
struct ith_type<I, tuple<T1, Ts...>> {
    using type = typename ith_type<I-1, tuple<Ts...>>::type;
};

template<typename T1, typename... Ts>
struct ith_type<0, tuple<T1, Ts...>> {
    using type = T1;
};
```

- Where in the code above is the loop?
- (Or: look-out for recursion!)
- How is the loop (i.e. recursion) terminated?
- (Or: look-out for specialisation!)

If you do a self study: Go back to where all this started (where the attempt was made to implement the global `get`), then once more go through all the intermediate steps up to here. If you think you grasped the principle, **close this page**, then implement `ith_type` from scratch, all on your own.

Example (cont.): my_tuple/demo.cpp

```
template<std::size_t I, typename T>
auto get(const T& arg)
    -> typename std::enable_if<I != 0,
                               typename ith_type<I, T>::type
                              >::type {
    return get<I-1>(arg.ms);
}
template<std::size_t I, typename T>
auto get(const T& arg)
    -> typename std::enable_if<I == 0,
                               typename ith_type<I, T>::type
                              >::type {
    return arg.m1;
}
```

- **Now: You are EXPECTED to understand this!**
- (Well, maybe you need to study it for some minutes … :-).)

As *Template Meta Programming* adopts the functional programming style, it could definitely pay to study a Language like *Haskell*. A particularly well written and amusing E-book on it is this: http://learnyouahaskell.com (Online-reading is even free!)

Optional Example (cont.): my_tuple/demo.cpp

```
template<typename... Ts>
auto make_tuple(Ts&&... args) -> tuple<Ts...> {
    return {std::forward<Ts&&...>(args...)};
}

std::string to_string(const tuple<> &) {
    return std::string{};
}

template<typename... Ts>
std::string to_string(const tuple<Ts...> &t) {
    std::ostringstream s;
    s << t.m1;
    if (sizeof...(Ts) > 1)
        s << ", " << to_string(t.ms);
    return s.str();
}
```

- Where above does the simply unpack a parameter pack?
- Where is "true" meta programming, i.e ...
    - ... repetition implemented by recursion?
    - ... with termination through specialisation?

Optional Example (cont.): my_tuple/demo.cpp

```
template<typename... Ts, std::size_t... Is>
auto to_string_helper(const tuple<Ts...> &t,
                      std::index_sequence<Is...> =
                      std::index_sequence_for<Ts...>{}) {
    std::ostringstream s;
    using left_to_right = int[];
    static_cast<void>(
        // only the side effects of the following are of interest
        left_to_right{0, // <-- 0 required in case of empty pack
            (void(s << (Is == 0? "" : ", ") << get<Is>(t)), 0)...
        }                                    // here "runs"    |||
                                             // the loop(!) ---^^^
    );
    return s.str();
}
template<typename... Ts>
inline auto to_string(const tuple<Ts...>& t) {
    return to_string_helper(t, std::index_sequence_for<Ts...>{});
}
```

- Which language feature makes the above dependant on C++14?
- Understand that the essential part of the code above is this:
  - `(`………`(s << (Is == 0? "" : ", ") << get<Is>(t))`………`)...`
  - with the ellipsis (`...`) appended to unpack the parameter pack.

Fold expressions

- C++1z generalises unpacking parameter packs
  - Syntactically this is called fold expressions
  - Avoids the need to depend on side effects of initialisation

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.) my_tuple/demo.cpp

```
template<typename... Ts, std::size_t... Is>
auto to_string_helper(const tuple<Ts...> &t,
                      std::index_sequence<Is...> =
                      std::index_sequence_for<Ts...>{}) {
    std::ostringstream s;
    ((s << (Is == 0? "" : ", ") << get<Is>(t)), ...);
    return s.str();
}
template<typename... Ts>
inline auto to_string(const tuple<Ts...>& t) {
    return to_string_helper(t, std::index_sequence_for<Ts...>{});
}
```

- Which language feature makes the above dependant on C++1z?
- Understand that the essential part of the code above is this:
    - `(s << (Is == 0? "" : ", ") << get<Is>(t)), ...`
    - with the ellipsis (`...`) appended to get the loop running.
    - Why is it once more parenthesized in the above code?
    - What is the terminating semicolon good for?

C++20 Concepts

- Concepts can be thought of as
  - Constraints for template arguments
  - A major intent is to improve error messages*

http://en.cppreference.com/w/cpp/concept

*: Currently, if a template instantiation fails, the root cause of the problem often is everything but clear to the compiler:
Usually the client (that instantiated the template) has created the problem by using a type that is not a model of the expected concept. But because the manifestation of the problem is in the instantiated template code – **frequently not authored by the client using it** – the error message gives no good idea what actually went wrong.
Furthermore – in an attempt to be helpful – the error message often reveals (too) much of the internal state the compiler had when discovering the problem and is more confusing as enlightening.

Basic Idea

- In a template argument list
    - Use name of a "concept" instead just `typename`
    - Define requirements for models of concept elsewhere
- In case of compile errors:
    - Blame client by referring to point of instantiation
    - Instead showing code from inside the template implementation

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH MICROCONSULT

Current State

- Still experimental* – available as compiler extension only:
    - http://www.generic-programming.org/software/ConceptGCC
    - http://www.generic-programming.org/software/ConceptClang
- Expertimental libraries:
    - http://www.generic-programming.org/software/libraries.php

*: The idea of concepts has been around for long, refined in many iterations.
Here are two talks on *Concepts Lite* as of *CppCon 2014*:
https://www.youtube.com/watch?v=qwXq5MqY2ZA
https://www.youtube.com/watch?v=NZeTAnW5LL0

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH