Templates

- a template can be either a class or a function
  - a *class template* defines a family of classes
  - a *function template* defines a family of functions
- *object-oriented* construct
  - static polymorphism, a.k.a. compile-time polymorphism
- powerful and useful in many ways
  - traditional generic data structure design – element type parameterization
  - an alternative to function overloading – can also be combined with function overloading
  - policy design – parametrizing e.g. storage policy, ordering policy, etc., for containers
  - template meta programming – compile time type classification, type property inspection, type translation, code selection, …
- pre-runtime
  - static type checking
  - highly optimizable
  - flexibility combined with safety and efficiency
- we will on the way see
  - programming techniques related to template programming
  - use of rvalue references, move semantics, perfect forwarding – all very important features of C++
  - core language features used in connection with templates – **auto**, **decltype**, **noexcept**, **constexpr**
  - examples of standard library components

Function templates

```
template<typename T>
const T& max(const T& x, const T& y)
{
   return x < y ? y : x;
}
```

- *template parameter list* – a comma separated list of *template parameters* within angle brackets

  template< *template parameters* >

- *template type parameter* – two alternatives, no semantic difference

  typename T   **class** T

- other kind of template parameters – *non-type parameter*, *template template parameter*, *template parameter pack*,...
- note – `max` is designed and implemented with care
  - the only requirement on T is **operator**<
  - there should also be an overloading with a second template parameter for a comparison predicate
  - pass by *reference* – T need not be copyable
  - pass by reference to **const** – *in semantics* enforced and arguments can be *constants* or *literals*
- be aware of special cases
  - wrong semantics if T is, e.g., a pointer type

Function template instantiation

- *template instantiation* – the act of instantiating a template

```
int a{ 1 };
int b{ 2 };

int c = max(a, b);        // name lookup…
```

- if no other, better candidate is found by name look up
  - the function call will create an **int** *instance* for our template `max` ()
  - such an instance is an *implicit specialization* for **int**
  - template argument for T is deduced from the type of the function call arguments, i.e. the type(s) for a and b
  - x and y have the same template type T – a and b must have same type (**int**)
  - no implicit type conversion of arguments is allowed in this context
- *explicit instantiation* is a *declaration* using the keyword **`template`** *only* in front of a declaration or definition

```
template const int& std::max<int>(const int&, const int&);
```

  - this will force the compiler to create *one* instance for the entire program
  - used to make, e.g., compilation more efficient
  - this is *not* an *explicit specialization* (next slide)
- how instances are managed is implementation dependent
  - the standard does not specify

Explicit function template specialization (1)

- the original declaration of a template is called ***primary template***

```
template<typename T> const T& max(const T& x, const T& y) { … }
```

  - the name for a primary is a simple *identifier* – `max`
- an ***explicit specialization*** is typically defined to handle a *special case*, e.g.
  - when the *template primary* does not work for a specific type
  - when a more efficient solution can be found for a specific type
- an explicit specialization is introduced by **`template`**`<…>`

```
template<> max<const char*>(const char*&, const char*&) { … }
```

  - specialization for **const char***
  - the name is a *template-id* – `max<const char*>`
  - is a ***full specialization*** if no template parameters are declared – **`template`**`<>`
  - otherwise it’s a ***partial specialization*** (examples will follow)
- a primary and its specializations are *separate* definitions
  - no code sharing

Explicit function template specialization (2)

- an ***explicit specialization*** for `std::max` for **const char**`*` would be

```
template<> const char*& max<const char*&>(const char*& x, const char*& y)
{
   return (strcmp(x, y) < 0) ? y : x;
}
```

  – T is replaced with **char*** (& must remain)

- don’t specialize a function template unless needed
  – specializations never participate in overload resolution
  – but, an existing specialization will be used, *if* its primary is chosen by overload resolution
- prefer to overload with an ordinary function

```
const char* max(const char* x, const char* y)
{
   return (strcmp(x, y) < 0) ? y : x;
}
```

  – obviously no point in specializing in this case
  – this function *will* participate in overload resolution, and will be chosen before any template is considered

Function templates and overload resolution

Function templates can be overloaded with both template functions and normal functions.

Overload resolution basically goes through the following steps to find a function to match a call:

1. if there is a normal function that exactly matches the call, that function is selected, *else*
2. if a function template can be instantiated to exactly match the call, that specialization is selected, *else*
3. if type conversion can be applied to the arguments, allowing a normal function to be used as a unique best match, that function is selected, *else*
4. overload resolution fails – either no function matches the call, or the call is ambiguous

```
template<typename T> const T& max(const T& x, const T& y);

int    a, b;
double x, y;
…

a = max(a, b);              // OK, a and b have same type, int

x = max(x, y);              // OK, x and y have same type, double

x = max(a, x);              // ERROR, a and x has different types

x = max<double>(a, x);      // explicit instantiation allows for implicit type conversion, a is converted to double
```


Non-type template parameters

```
template<typename T, std::size_t N>
T* begin(T (&a)[N])
{
   return a;
}

template<typename T, std::size_t N>
T* end(T (&a)[N])
{
   return a + N;
}


int a[]{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };                  // type is int[9]

for (auto it = begin(a), it != end(a); ++it)
   cout << *it << endl;
```

- N is a *non-type template parameter*
  - acts as a *constant* within the template
  - the dimension of the array will be deduced from the function call argument
- it is recommended to pass arrays to functions by reference, instead as by pointer

Examples taken from the standard library, *range access*.

*Function templates, **auto** and **decltype***

```
template<class Container>
auto begin(Container& c) -> decltype(c.begin())          // also for const Container&
{
   return c.begin();
}

template<class Container>
auto end(Container& c) -> decltype(c.end()))             // also for const Container&
{
   return c.end();
}


vector<int> v{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };

for (auto it = begin(v), it != end(v); ++it)
   cout << *it << endl;
```

- return type is not possible to determine from the template parameter
  - the actual type for `Container` must be known, and then what the member functions `begin()` and `end()` returns
- **decltype** is an *unevaluated context* – c.begin() and c.end() are *not* invoked
- In C++14 **auto** will be sufficient in cases where the return type can be deduced from the code in the body (as in these examples)

Note: This syntax is a reminiscent of the lambda syntax – [*capture*] (*parameter-list*) -> *return-type* { *statements* }

Class templates

```
template<class T1, class T2> struct pair
{
   typedef T1 first_type;
   typedef T2 second_type;

   T1 first;
   T2 second;

   pair(const pair&) = default;
   pair(pair&&) = default;
   constexpr pair() : first(), second() {}
   pair(const T1& x, const T2& y) : first(x), second(y) {}
   // more constructors

   pair& operator=(const pair& p);
   pair& operator=(pair&& p);
   // more assignment operators

   void swap(pair& p)
     noexcept(noexcept(swap(first, p.first)) && noexcept(swap(second, p.second))) { … }
};

template<class T1, class T2>
void
swap(pair<T1, T2>& x, pair<T1, T2>& y) noexcept(noexcept(x.swap(y))) { x.swap(y); }
```


Comments on std::pair

```
pair<int, string>  p{ 4711, "Eau de Cologne" };

pair<int, string>::first_type  x = p.first;
```

- two template *type parameters*, `T1` and `T2`
- *member types* `first_type` and `second_type`
  - `T1` and `T2` – but with better names – are made available to code using `pair`
  - common practice for class templates
- data members `first` and `second`
  - direct access causes no problem – **public**
- `pair` means `pair<T1, T2>`, when used within the template
- **`constexpr`** `pair()` means that a call to the constructor is a *constant expression*
  - can be evaluated during compile time
  - default initialized `pair` objects can be used in static contexts
- the *noexcept specification* for member `swap` specifies that it will not throw
  - *if* swapping `T1` objects will not throw, *and*
  - *if* swapping `T2` objects will not throw
- the *noexcept specification* for non-member `swap` specifies that it's no-throw, if member `swap` is no-throw

Note: **noexcept** is an operator – **noexcept**(*expression*)

Helper function make_pair()

```
template<class T1, class T2>
pair<T1, T2>
make_pair(T1&& x, T2&& y)
{
   return pair<T1, T2>(std::forward(x), std::forward(y));
}

auto p = make_pair(4711, string{"Eau de Cologne"});               // pair<int, string>
```

- common practice to have a helper function to create an instances of a class template for given arguments
  - creating a pair object becomes easy
  - the types of the template arguments `T1` and `T2` are deduced from the given call arguments
  - reference collapsing is applied to T1&& and T2&& (more about that later)
- `std::forward()` is a utility function for ***perfect forwarding***
  - makes it work as if the call to the pair constructor was made in the client code
  - preserves the actual type category of the arguments – lvalue or rvalue
- if the first argument is an lvalue of type `L` and the second argument is an rvalue of type `R`, the signature will effectively be

```
pair<L, R> make_pair(L& x, R&& y);
```

  - why cannot `std::move()` be used in this context?

Class template instantiation and specialization

- ***implicit instantiation*** occurs when the context requires an instance of a class template
  - class template arguments are *never deduced*, but default arguments can be used

```
template<class T, class Allocator = allocator<T>> class vector;

vector<int> v;          // definition of v require instance of std::vector for int
```

  - class template member functions are *instantiated when called*

```
v.push_back(x);
```

- an ***explicit instantiation*** is a declaration using the keyword **`template`**

```
template class vector<int>;
```

- a class template ***specialization*** is introduced by **`template`**`<…>` – the name is a *template-id*

```
template<typename Allocator> class vector<bool, Allocator> { … };
```

  - specialization for vector<**bool**>
  - this is a ***partial specialization***, since there is (at least) one template parameter left (`Allocator`)
  - a ***full specialization*** is introduced by **`template`**`<>`

```
template<> class vector<bool, std::allocator<bool>> { … };
```

  - primary and specialization have separate definitions – no code sharing

Default template parameter arguments

```
template<typename T = int,                                 // for demonstration only
         typename U = std::allocator<T>,
         template<typename = T, typename = U> class Container = std::vector>
ostream& operator<<(ostream& os, const Container<T, U>& c);

template<size_t N = 32> bitset;
```

– Note: only `std::allocator<T>` is in practice realistic to have as a default argument

- if all template parameters have a default argument, the template argument list can be empty

```
template<typename T = int> class X { … };

X<> x1;          // OK, same as X<int>
X   x2;          // Syntax error
```

- ambiguity between a *type identifier* (*type-id*) and an *expression* is resolved to a *type identifier*

```
template<typename T> void fun();      // function template with a type parameter
template<int I>      void fun();      // function template with a non-type parameter

void g() { fun<int()>(); }            // int() is resolved as a type-id – first fun() is called
```

– what type?

Separate definitions for members of a class template

```
template<typename T>
struct S
{
   T t_{ T() };                              // non-static data member definition

   T get_t();                                // non-static member function declaration

   static T c_;                              // static data member declaration

   static T get_c();                         // static member function declaration
};

template<typename T>                         // non-static member function definition
T S<T>::get_t() { return t_; }

template<typename T>                         // static data member definition
T S<T>::c_{ T() };

template<typename T>                         // static member function definition
T S<T>::get_c() { return c_; }
```

- *instantiation argument* used in qualified names – `S<T>::`*name*
- expression `T()` to initialize data members of a generic type `T`

Variadic templates

- a ***template parameter pack*** is a template parameter that accepts zero or more template arguments.

```
template<class... Types>
class tuple { … };
```

  – Types is a template parameter pack

```
tuple<>              t0;                    // Types contains no arguments
tuple<int>           t1{ 17 };              // Types contains one arguments, int
tuple<int, double> t2{ 17, 3.14 };        // Types contains two arguments, int and double
```

- a ***function parameter pack*** is a function parameter that accepts zero or more arguments

```
template<class... Types>
void fun(Types... args);                    // function parameter pack
```

  – `args` is a function parameter pack

```
fun();                 // args contains no arguments
fun(1);                // args contains one argument, type int
fun(2, 3.14);          // args contains two arguments, type int and double
```

Variadic print function

```
template<typename T> void print(const &T x)
{
   cout << x;
}

template<typename First, typename... Rest>
void print(First first, Rest... rest)                  // Rest is expanded
{
   cout << first << ’ ’;
   print(rest...);                                     // rest is expanded
}
```

- `Rest` is a template parameter pack
- `rest` is a function parameter pack
- `rest...` is a *pack expansion* – consists of a pattern and an ellipsis – `rest` is the *pattern*
  - beside *expanding*, the only thing one can do with a parameter pack is to take its *size* with **`sizeof...`**

```
print();                             // error: no function matches call print()

print(17);                           // 17
print(17, "Hello");                  // 17 Hello
print(17, "Hello", 3.14);            // 17 Hello 3.14
print(17, "Hello", 3.14, ’X’);       // 17 Hello 3.14 X
```

Variadic emplace functions

- the purpose is to avoid copies – for example when inserting into a container and we have suitable constructor arguments at hand
- pass constructor arguments instead of ready-made objects – by rvalue reference or by lvalue reference
- objects are created in-place by a matching constructor – move or copy

```
template<typename T> class Container
{
public:
   void insert(const T& x) { … }

   template<typename... Args>
   void emplace(Args&&... args) { construct(&storage_, std::forward<Args>(args)...); }

private:
   template<typename... Args>
   void construct(T* p, Args&&... args) { new(p) T{ std::forward<Args>(args)... }; }

   T storage_;        // very, very simplified storage…
};

Container<pair<int, string>> c;

string s{ "foobar" };

c.emplace(4711, s);                 // note: first argument is an rvalue, second is an lvalue

c.insert(make_pair(4711, s));       // this is what we avoided, first creating a pair and then insert it
```

Template parameter overview (1)

- *template type parameter* (**typename** or **class**)

```
template<typename T> class X;
```

  – actual T can be any type (fulfilling the requirements the implementation of X puts on T)

- *template non-type parameter* – shall have one of the following types:
  – *integral* or *enumeration* type

```
template<int n, Colour c> …
```

  – *pointer* to object, or to function

```
template<double* p, void (*pf)()> …
```

  – *lvalue reference* to object or to function

```
template<X& r, void (&rf)()> …
```

  – *pointer to data member, or member function*

```
template<int X::*pm, void (X::*pmf)()> …
```

  – `std::nullptr_t` – hard to find examples...

Template parameter overview (2)

- *template template parameter*

```
template<typename T, typename U, template<typename, typename> class Container>
ostream& operator<<(ostream& os, const Container<T, U>& c)
{
   copy(begin(c), end(c), ostream_iterator<T>(os, " "));
   return os;
}

vector<int> v{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };

cout << v << endl;
```

  – `Container` can be instantiated by any class type having two type parameters

```
template<typename, typename> class Container;
```

  – `vector`, and all other sequence containers, except `array`, have two template type parameters

```
template <class T, class Allocator = allocator<T>> vector;
```

  – default arguments for template parameters are *not* considered in this context

Template parameter overview (3)

- *template type parameter pack*

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

  – `tuple` can be instantiated with zero or more type arguments

```
tuple<int, string, double> fragrance{ 4711, "Eau de Cologne", 39.39 };
```

  – the **`sizeof...`** operator yields the number of arguments provided for a parameter pack

```
static const std::size_t number = sizeof...(Types);
```

- *template non-type parameter pack*

```
template<typename T, int... Dims> class Multi_Array;
```

  – `Multi_Array` can be instantiated with a type and zero or more **int** values

```
Multi_Array<string, 3, 4> m34;
```

Template parameter overview (4)

- a template parameter (T) can be used in declaration of subsequent template parameters and their default arguments

```
template<class T, T* p, class U = T> class X;
```

- a template parameter of type *array of T* is adjusted to *pointer to T* by the compiler

`template<int a[10]>` *is adjusted to* `template<int* a>`

  – as for an ordinary function parameter of array type

- a template parameter of type *function* is adjusted to *pointer to function* by the compiler

`template<int f()>` *is adjusted to* `template<int (*f)()>`

  – as for an ordinary function parameter of function type

The two, in practice, dominant kinds of template parameters are

- *type parameters*
- *integral non-type parameters*

Type equivalence

Two template identifiers refer to the same class or function, if their

- names are identical
- type arguments are the same types
- non-type arguments of integral or enumeration type have identical values
- non-type arguments of pointer or reference type points/refer to the same object or function
- template arguments refer to the same template

```
template<typename T, T t, T* p, template<typename> class C> class X;

template<typename T> class Y;
template<typename T> class Z;

int a;
int b;

X<int,  100, &a, Y>  x1;

X<int, 2*50, &a, Y>  x2;          // x1 and x2 have same type

X<int,   10, &b, Z>  x3;          // x3 have not same type as x1 and x2
```

Member templates

```
template<class T1, class T2>
struct pair
{
   …
   template<class U, class V> pair(U&& x, V&& y);
   …
};
```

- a class template or function template can be declared as member of a class
  - such a template is called a ***member template***
  - a member function template cannot be virtual and never override a virtual function
  - allowing for type conversions is one use – if `U` is convertible to `T1`, and `V` is convertible to `T2`, the constructor above is fine
- copy/move constructors and copy/move assignment operators are *not* templates
- separate member template declaration syntax

```
template<class T1, class T1>          // class template parameters
template<class U, class V>            // member template parameters
pair<T1, T2>::pair(U&& x, V&& y)
   : first{ std::forward<U>(x) }, second{ std::forward<V>(y) }
{}
```

Alias templates

The declaration for `std::vector` is:

```
template<class T, class Allocator = std::allocator<T>> class vector;
```

- an ***alias template*** for vector could be

```
template<typename T>
using default_alloc_vector = std::vector<T, std::allocator<T>>;
```

  - template version of the alias declaration
  - implicit partial instantiation of Allocator
  - can be used as

```
default_alloc_vector<int> v;
```

  - which is the same as

```
vector<int, std::allocator<int>> v;
```

- `default_alloc_vector` can be used as an argument for a template template type parameter with *one* parameter

```
template<typename T, template<typename> class C = default_alloc_vector> class X {};

X<int> x;
```

  - vector cannot, since it has two template parameters

Dependent names

Inside a template, constructs have semantics which may differ for different instances.

- such constructs *depends* on the template parameters
- in particular *types* and *expressions* may depend on the type and/or the value of template parameters, which are determined by the template arguments

Some examples:

```
template <typename T>
struct Derived : public Base<T>             // Base<T> depend on T
{
   typename T::X* x_ptr;                    // T::X depend on T

   int value = Base<T>::value;              // Base<T>::value depend on T

   void memfun()
   {
      typename vector<T>::iterator it;      // vector<T>::iterator depend on T
      …
   }
};
```

- dependent names are by definition *not* assumed to name a type, unless qualified by **typename**
  - `T::X` could be the name of a data member, e.g.

SFINAE – Substitution Failure Is Not An Error

Invalid substitution of template parameters is *not* in itself an error.

When the compiler is deducing the template parameters for a function template from a function call

- if the deduced parameters would make the signature invalid, that function template is not considered for overload resolution
  - it does not stop the compilation with a compile error

Example – the `std::enable_if` template, from `<type_traits>`

- primary template have no members

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

- partial specialization for **true** have a member **typedef** for `T`, named `type`

```
template<typename T> struct enable_if<true, T> { typedef T type; };
```

- typically used in combination with some type traits component from the standard library

```
template<typename T>
typename std::enable_if< std::is_pod<T>, void >::type
fun( … )
{
    …
}
```

  - if `T` is *not* a POD there is no member `type` and this overload of `fun` is then *discarded* – it is *not an error*

SFINAE – example of practical use

- if T is POD, use std::memcpy to copy T objects – efficient byte-by-byte copy

```
template<typename T>
typename std::enable_if< std::is_pod<T>::value, void >::type
copy(const T* source, T* destination, unsigned count)
{
   std::memcpy(destination, source, count * sizeof(T));
}
```

  - if T is POD, `std::is_pod<T>::value` will be **true**
  - `std::enable_if<std::is_pod<T>::value, void>::type` will be defined (**void**), otherwise not (failure)
  - if `type` is defined the overload above will be enabled (and the one below will be disabled)

- if T is *non*-POD, use object-by-object copy assignment for T

```
template<typename T>
typename std::enable_if< !std::is_pod<T>::value, void >::type
copy(const T* source, T* destination, unsigned count)
{
   for (unsigned i = 0; i < count; ++i)
   {
      *destination++ = *source++;
   }
}
```

Utility function std::move() (1)

Typically used to allow for move semantics on lvalues – example from the standard library:

```
template<class T>
void swap(T& x, T& y)
   noexcept(is_nothrow_move_constructible<T>::value &&
            is_nothrow_move_assignable<T>::value)
{
   T tmp{ std::move(x) };
   x = std::move(y);
   y = std::move(tmp);
}
```

- x and y are lvalues
- in this context move semantics is appropriate to exchange the values of x and y
- std::move() converts its argument to a nameless rvalue reference (T&&)
  - if T have move constructor and move assignment operator they will be used
  - if not, the copy constructor and copy assignment operator will be used
- whether or not swap may throw an exception is specified by two *type property traits*
  - `value` will be **true** if `T` objects can be *move copied* and *move assigned* without throwing

Utility function std::move() (2)

```
template<typename T>
typename std::remove_reference<T>::type&&
move(T&& t)
{
   return static_cast<typename std::remove_reference<T>::type&&>(t);
}
```

Much to learn from this simple function template…

- move() uses *type modifier traits* class std::remove_reference:

```
template<typename T> struct remove_reference { typedef T type; };          // primary

template<typename T> struct remove_reference<T&> { typedef T type; };      // spec 1

template<typename T> struct remove_reference<T&&> { typedef T type; };     // spec 2
```

  - the primary template matches any type – used for non-reference types
  - the first specialization matches *lvalue references* (T&)
  - the second specialization matches *rvalue references* (T&&)
  - `std::remove_reference<T>::type` will always be T
- the return type of move() will always be rvalue reference (T&&)
- the type conversion will effectively be `static_cast<T&&>(t)`
- `std::remove_reference<T>::type` is a dependent name – **typename** required

Utility function std::move() (3)

Reference collapsing rules

A&	&	*becomes*	A&	(1)
A&	&&	*becomes*	A&	(2)
A&&	&	*becomes*	A&	(3)
A&&	&&	*becomes*	A&&	(4)

Special template argument deduction rules for function templates

```
template<typename T>
void foo(T&& arg);        // deduced parameter type – type deduction takes place – && = universal reference
```

If foo() is called with an

- *lvalue* of type A, T resolves to A&, and by reference collapsing rule 2 the type of `arg` effectively becomes **A&**
- *rvalue* of type A, T resolves to A and the type of `arg` becomes **A&&** (no reference collapsing involved)

So, if std::move() is called with an

- *lvalue* of type A, T resolves to A& and the parameter type (A& &&) becomes A& – `move(A& t)`
  - *spec 1* of std::remove_reference is used – `std::remove_reference<A&>`
- *rvalue* of type A, T resolves to A and parameter type becomes A&& – `move(A&& t)`
  - *primary* of std::remove_reference is used – `std::remove_reference<A>`
- cast is always to rvalue reference, A&&

Note: Reference collapsing rule 1 has always been in effect since C++98.

Perfect forwarding

Rvalue references was also designed to solve the *perfect forwarding problem*.

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg&& arg)
{
   return shared_ptr<T>{ new T{ arg } };    // arg is an lvalue (the name rule)
}
```

- we want this to work as if the argument passed to factory() was passed directly to the T constructor
  - lvalue as lvalue
  - rvalue as rvalue
- std::forward() does this

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg&& arg)
{
   return shared_ptr<T>{ new T{ std::forward<Arg>(arg) } };
}
```

- if factory() is called with an *lvalue* of type X, Arg resolves to X& and the type for parameter arg will be **X&** (X& &&)
  - forward will be instantiated `std::forward<X&>(arg)`
- in factory() is called with an *rvalue* of type X, Arg resolves to X and the type for parameter arg will be **X&&**
  - forward will be instantiated `std::forward<X>(arg)`

Utility function std::forward()

Lvalue version:

```
template<typename T>
constexpr T&& forward(typename std::remove_reference<T>::type& t)
{
   return static_cast<T&&>(t);
}
```

- T&& effectively becomes T& (template argument deduction rule and reference collapsing rule)
- return type will be T& – conversion will be **static_cast**<T&> – parameter is always T&

Rvalue version:

```
template<typename T>
constexpr T&& forward(typename std::remove_reference<T>::type&& t)
{
   static_assert(!std::is_lvalue_reference<T>::value,
                 "template argument substituting T is an lvalue reference type");
   return static_cast<T&&>(t);
}
```

- T&& will be T&& (template argument deduction rule, no reference collapsing involved)
- if instantiated with an lvalue reference type the program is ill-formed – prevent strange things like

```
std::forward<string&>( string{} )    // string{} creates a prvalue
```


Policy design (1)

Policy design is about letting a feature, a *policy*, of a type, the *host*, be a template type parameter.

- Two copy policies for objects of type T:
  - deep copy

```
template<typename T>
struct deep_copy
{
   static void copy(T*& destination, T*& source)
   {
      destination = new T{ *source };
   }
};
```

  - destructive copy (move semantics)

```
template<typename T>
struct destructive_copy
{
   static void copy(T*& destination, T*& source)
   {
      destination = source;
      source = nullptr;
   }
};
```

Note: Alternatively the functions could be templates, instead of the policy classes – actually a better choice!

Policy design (2)

A smart pointer class with a copy policy – `deep_copy` as default

```
template<typename T, template<typename T> class copier = deep_copy>
class smart_ptr : private copier<T>
{
   public:
      smart_ptr(T* p) : ptr_{ p } {}

      ~smart_ptr() { delete ptr_; }

      smart_ptr(smart_ptr& value) { this->copy(ptr_, value.ptr_); }   // this-> required

      smart_ptr& operator=(smart_ptr& rhs)
      {
         T* tmp_ptr;
         copier<T>::copy(tmp_ptr, rhs.ptr_);                          // or, alternatively
         delete ptr_;
         ptr_ = tmp_ptr;
         return *this;
      }
      …

   private:
      T* ptr_;
};
```

Note: Names are not looked up in dependent base classes – call to copy() require qualification of some kind.

Policy design (3)

A better alternative, in this case, would be to make member functions templates, instead of policy classes.

```
struct deep_copy
{
   template<typename T>
   static void copy(T*& destination, T*& source)
   {
      destination = new T{ *source };
   }
};

struct destructive_copy
{
   template<typename T>
   static void copy(T*& destination, T*& source)
   {
      destination = source;
      source = nullptr;
   }
};
```

- by this we can take advantage of template function argument deduction
- we will also see some other simplifications and also generalizations

Policy design (4)

```
template<typename T, class copier = deep_copy>                        // note!
class smart_ptr                                                       // note!
{
   public:
      smart_ptr(T* p) : ptr_{ p }{}

      ~smart_ptr() { delete ptr_; }

      smart_ptr(smart_ptr& value) { copier::copy(ptr_, value.ptr_); }    // note!

      smart_ptr& operator=(smart_ptr& rhs)
      {
         T* tmp_ptr;
         copier::copy(tmp_ptr, rhs.ptr_);                              // note!
         delete ptr_;
         ptr_ = tmp_ptr;
         return *this;
      }
      …

   private:
      T* ptr_;
};
```

Meta programming and type traits

Meta programming is about programs that create or modifies programs.

- the template instantiation process in C++ can be used as a *computational engine – template meta programming*
- parametrized types and (compile-time) constants can be used to record *state*
- template specializations can be used to implement *conditions*
- C++ is both the *metalanguage* and the *object language*

The standard library define components to be used by C++ programs, particularly in templates, to

- support the widest possible range of types
- optimize template code usage
- detect type related user errors
- perform type inference and transformation at compile time

It includes:

- type *classification* traits
  - describe a complete taxonomy of all possible C++ types, and state where in that taxonomy a given type belongs
- type *property inspection* traits
  - allow important characteristics of types or of combinations of types to be inspected
- type *transformations*
  - allow certain properties of types to be manipulated

Type classification traits (1)

- *unary type traits* describes a property of a type
  - *type categories* – is_integral, is_floating_point, is_array, is_pointer, is_class, is_function, is_reference, is_arithmetic, is_fundamental, is_object, is_scalar, is_compound, is_member_pointer, …
  - *type properties* – is_const, is_polymorphic, is_abstract, is_signed, is_copy_constructible, has_virtual_destructor, …
  - *type property queries* – alignment_of, …
- *binary type traits* describes a relationship between two types
  - is_same, is_base_of, is_convertible
- *transformation traits* modifies a property of a type
  - const-volatile modifications – e.g. remove_const, remove_cv, add_volatile
  - reference modifications – remove_reference, add_rlvalue_reference, add_rvalue_reference
  - sign modifications – make_signed, make_unsigned
  - array modifications – remove_extent, remove_all_extent
  - pointer modifications – remove_pointer, add_pointer
  - other transformations – …

(There is about 90 type traits in C++11).

Type classification traits (2)

- class template `integral_constant` is the common base class for all value-based type traits.

```
template<typename T, T v>
struct integral_constant
{
   static constexpr T value{ v };                          // declaration and initialization
   using value_type = T;
   using type = integral_constant<T, v>;                   // the type itself
   constexpr operator value_type() { return value; }
};

template<typename T, T v>
constexpr T integral_constant<T, v>::value;               // definition for value
```

- `true_type` and `false_type` are provided for convenience – most value traits are Boolean properties and will inherit from these

```
using true_type = integral_constant<bool, true>;

using false_type = integral_constant<bool, false>;
```

- `is_array` as an example

```
template<typename> struct is_array : public false_type {};                // primary

template<typename T, std::size_t Size> struct is_array<T[Size]> : public true_type {};

template<typename T> struct is_array<T[]> : public true_type {};
```

Type classification traits (3)

Examples of some simple type traits.

```
template<typename> struct is_const : public false_type {};

template<typename T> struct is_const<T const> : public true_type {};


template<typename T> struct remove_const { typedef T type; };

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


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


template<typename, typename> struct is_same : public false_type {};

template<typename T> struct is_same<T, T> : public true_type {};


template<typename T> struct is_arithmetic
  : public integral_constant<bool, (is_integral<T>::value || is_floating_point<T>::value)> {};


template<typename T> struct is_object
  : public integral_constant<bool, !(is_function<T>::value
                                  || is_reference<T>::value
                                  || is_void<T>::value)> {};
```


Example: Dispatching constructor

Example from the containers library – wrong constructor can be chosen…

```
template <typename T> class Container {
    …
    Container(size_t n, const T& value) : size_(n), data_(allocate(size_))
    { std::fill_n(begin(), n, value); }

    template <typename InputIterator>
    Container(InputIterator first, InputIterator last) {
        typedef typename std::is_integral<InputIterator>::type Integral;   // check type category
        initialize_dispatch(first, last, Integral());                      // dispatch accordingly
    }

    template <typename Integer>                       // corresponds to the first constructor
    void initialize_dispatch(Integer n, Integer value, std::true_type)
    { std::fill_n(begin(), n, value); }

    template <typename InputIterator>                 // corresponds to the second constructor
    void initialize_dispatch(InputIterator first, InputIterator last, std::false_type)
    { std::copy(first, last, begin()); }              // memory must be allocated first!

};
```

- `size_t` is an **unsigned** integer type
- if `T` is, e.g. **int**, an instance of the template constructor (aimed for iterators!) is a better match than the first constructor
- check what type category InputIterator in the second constructor belongs to and dispatch to the corresponding helper function

Example: Select the most efficient implementation of several possible (1)

The standard library divide iterators into different categories – one important issue is how they can be moved:

- an *InputIterator* can be moved forward one element at the time with `++`
- a *BidirectionalIterator* can be moved forward one element at the time with `++` and backward one element at the time with `--`
- a *RandomAccessIterator* can be moved also by pointer arithmetics, i.e., also by using `+`, `+=`, `-`, and `-=`

advance() is a utility function to move an iterator a specified number of steps – `advance(it, n)`

- for an InputIterator n must be non-negative and the advancement is done by repeating `++`it n times
- for a BidirectionalIterator advancement is done by repeating `++`it n times, if n >= 0, and repeating `--`it n times, if n < 0
- for a RandomAccessIterator advancement can be done with it `+=` n (n can be negative)
- advance() is implemented to select the most efficient way to advance an iterator, depending on which category it belongs to

```
template <typename InputIterator, typename Distance>
inline void advance(InputIterator& i, Distance n)
{
   typename iterator_traits<InputIterator>::difference_type d = n;
   advance_dispatch(i, d, typename iterator_traits<InputIterator>::iterator_category());
}
```

  - the name InputIterator is used to specify the minimum requirements for the iterator
  - Distance could be any relevant type – d is used to convert n to the correct difference type for the iterator in question
  - iterator_traits<InputIterator>::iterator_category gives the category type for InputIterator, e.g. random_access_iterator_tag
  - iterator_traits<InputIterator>::iterator_category() creates an object of that type – used to dispatch to a suitable overloading of advance_dispatch()

Example: Select the most efficient implementation of several possible (2)

- the basic requirement is that the given iterator must fulfill the requirements of InputIterator, and n >= 0 – dispatch to

```
template <typename InputIterator, typename Distance>
void advance_dispatch(InputIterator& i, Distance n, input_iterator_tag)
{
   while (n--) ++i;
}
```

- if the given iterator is a BidirectionalIterator, n is allowed to be negative – dispatch to

```
template <typename BidirectionalIterator, typename Distance>
void advance_dispatch(BidirectionalIterator& i, Distance n, bidirectional_iterator_tag)
{
   if (n > 0)
      while (n--) ++i;
   else
      while (n++) --i;
}
```

- if the given iterator is a RandomAccessIterator the iterator can be advanced in a more efficient way – dispatch to

```
template <typename RandomAccessIterator, typename Distance>
void advance_dispatch(RandomAccessIterator& i, Distance n, random_access_iterator_tag)
{
   i += n;
}
```

Example: Constraints check (1)

Suppose a class template `C` require a template type `T` to have a member function clone().

```
template<typename T>          // T must have member function T* clone() const;
class C
{
   …
private:
   template <T* (T::*)() const> struct _check_Cloneable {};

   using _check_Cloneable_t = _check_Cloneable<&T::clone>;
   …
};
```

- `_check_Cloneable` is a utility template class
  - a type definition is a *static* entity
  - `_check_Cloneable` is a primary with one member function pointer parameter (name left out)
- `_check_Cloneable_t` is declared to be a name for the specialization `_check_Cloneable<&T::clone>`
  - an alias declaration is a *static* entity
  - `T::clone` is looked up in T and checked *statically* – this is what we actually want to do!
  - if no such member function this will fail and generate a compile error

Note, leading underscores are used to emphasise internal use (typical compiler convention)

Example: Constraints check (2)

As an alternative, define a *concept class* for checking cloneability.

```
template<typename T>
struct _Cloneable
{
   void __constraints() { T* (T::*test)() const = &T::clone; }
};

template<typename T>
class C
{
   …
private:
   using __func = void (_Cloneable<T>::*)();                // simplifies the next declaration

   template<__func> struct _check_Cloneable {};

   using _check_Cloneable_t = _check_Cloneable<&_Cloneable<T>::__constraints>;
   …
};
```

- becomes more general
  - `_Cloneable` can be replaced by any concept class having member function `__constraints()`
  - `__constraints()` can have several checks
  - now we just have to find a way to simplify generating these declarations to check a concept

Example: Constraints check (3)

(*The simplification becomes a bit complicated though…*)

- a preprocessor function macro can be used to create the declarations to check a concept `_concept` for a type `_type`

```
#define CLASS_REQUIRES(_type, _concept) \
using __func_##_type##_concept = void (_concept<_type>::*)(); \
template <__func_##_type##_concept> struct _check_##_type##_concept {}; \
using _check_typedef_##_type##_concept = \
_check_##_type##_concept<&_concept<_type>::__constraints>
```

- simple use

```
CLASS_REQUIRES(T, _Cloneable);
```

  – all instances of `##_type` are replaced by `T`
  – all instances of `##_concept` are replaced by `_Cloneable`

- result (actually as a single line)

```
using __func_T_Cloneable = void (_Cloneable<T>::*)();

template <__func_T_Cloneable> struct _check_T_Cloneable {};

using _check_typedef_T_Cloneable = _check_T_Cloneable<&_Cloneable<T>::__constraints>;
```

- leading underscores in names are to be regarded as reserved for the implementation, don’t use for user names

Template compilation models

The *inclusion compilation model*

- the definition for a template is included in every file in which the template is instantiated
- one can still have a template function declaration on an inclusion file (fun.h) and its definition on a separate file (fun.tcc)
  - in such case the .h files pulls in the .tcc file at its end – #include "fun.tcc"
  - the alternative would be to have only the template function definition in fun.h, and no fun.tcc file
- a variation of this allows you to exclude #include "fun.tcc" in the fun.h file
  - in this case the compiler implicitly knows by some rule where to look for fun.tcc
- the inclusion compilation model is the model that most current C++ compilers provide

The *separation compilation model*

- basically the traditional model with header files and corresponding implementation files
  - an implementation file is *not* pulled into its header file
  - compiled separately
  - difficult to implement, few compilers have
- this compilation model is related to the keyword **export**
  - **export** is removed from C++11 – “ The **export** keyword is unused but is reserved for future use.”
  - seems meaningless to get into any details…

