Effective C++ Programming

NIE-EPC (v. 2021):

TEMPLATES, TEMPLATE INSTANTIATION, TEMPLATE ARGUMENT DEDUCTION, TEMPLATE SPECIALIZATION, VARIADIC TEMPLATES, TEMPLATE PARAMETER PACK

Templates — templated entities

- *Templates* = parametrized *generators of source code entities*.
- Possible *template-generated entities*:
  - *classes*,
  - *functions* (both free and member),
  - *variables* (since C++17),
  - *types* (*type aliases*; since C++11),
  - *concepts* (since C++20).
- *Template definition* = prescription on how the entity should be generated (with respect to template arguments).
  - Its generation is called "*instantiation*".
  - Resulting entity is called "*instance*" of a template.
- *Example:*
  - `wrapper` = *class template*, `wrapper<int>` = its instance = *class*.

```
template <typename T>
struct wrapper {
  T obj;
};
```

Templates — parameters and arguments

- Templates = parametrized generators of *source code* entities.
- *Implication — generation/instantiation* must happen *at compile time*.
  - It effectively generates source code that needs to be further compiled by a compiler.
  - *Note* — at *runtime*, templates no more exist (only their instances do).
- Generation of source code entities is parametrized by *template parameters*.
  - Arguments for these parameters — *template arguments* — need to be resolvable at compile time.
  - ⇒ Basically, there are two kinds of template parameters:
    - *type template parameters* — their arguments are types,
    - *non-type template parameters* — their arguments are compile-time constants; mostly *integral/Boolean*).
- *Example:*

```
template <typename T, size_t N>  // T - type template param., N - non-type template param.
struct Array {
  T data[N];
};
```

Templates — explicit instantiation

- Templates can be instantiated either *explicitly* or *implicitly*.
- *Explicit instantiation* forces a compiler to generate template instance without its usage.
  - *Example — source code…*

```
template <typename T> bool nonzero(T t) { return t != 0; }  // function template definition

template bool nonzero<int>(int);    // explicit instantiation of function nonzero<int>
template bool nonzero<long>(long);  // explicit instantiation of function nonzero<long>
```

  - …effectively *equivalent source code* after instantiation…

```
bool nonzero<int>(int t) { return t != 0; }
bool nonzero<long>(long t) { return t != 0; }
```

  - …resulting *machine code*:

```
bool nonzero<int>(int):
  test  edi, edi
  setne al
  ret

bool nonzero<long>(long):
  test  rdi, rdi
  setne al
  ret
```

- *Note* — explicit template instantiation is rarely used in practice.

Templates — implicit instantiation

- *Implicit instantiation* forces a compiler to generate template instance by using the template.
  - *Example — source code*…

```
template <typename T> bool nonzero(T t) { return t != 0; }  // function template definition

int main() {
  std::cout << (void*)nonzero<int>;   // print address of function nonzero<int>
}
```

  - …effectively *equivalent source code* after instantiation…

```
bool nonzero<int>(int t) { return t != 0; }

int main() {
  std::cout << (void*)nonzero<int>;
};
```

  - …resulting *machine code*:

```
bool nonzero<int>(int):
  test  edi, edi
  setne al
  ret

main:
  ...
```

Templates — header files

- *Implicit instantiation* is initiated by using the template.
- For instantiation, *template definition* needs to be available.
- *Implication* — if the template is not instantiated explicitly, its definition needs to be available in each translation unit where that template is used.
- *Typical solution of this problem* = putting definitions of templates into *header files*.
- *Note:*
  - In case of *function templates*, this can effectively result in the presence of a machine code of its instance — function — in multiple *translation units = object files*.
  - With templates, this does not break ODR (*one definition rule*).

Template argument deduction

- In *all cases*, template arguments may be provided explicitly.
- In *some cases*, they may be implicitly *deduced* by a compiler.
- Such *deduction* is mostly based:
  - either on *function arguments* for function templates,
  - or on initialization = *constructor arguments* for class templates (since C++17).
- ⇒ For deduction, template parameter needs to be somehow involved in the form of a function (constructor) parameter.
- *Example:*

```
template <typename T> bool nonzero(T t) { return t != 0; }

int main(int argc, char* argv[]) {
  std::cout << nonzero<int>(argc);  // explicitly-provided template argument
  std::cout << nonzero(argc);       // template argument deduced by function argument
}
```

  - ...effectively *equivalent source code* after instantiation...

```
bool nonzero<int>(int t) { return t != 0; }

int main(int argc, char* argv[]) {
  std::cout << nonzero<int>(argc);
  std::cout << nonzero<int>(argc);
}
```

Template argument deduction *(cont.)*

- Deduction of template arguments must be *unambiguous*.
- Otherwise, its resolution cannot take place.
- *Example:*

```
template <typename T>
bool larger(T a, T b) { return a > b; }

int main() {
  std::cout << larger(1, 2.0);
  // error: no matching function to call 'larger(int, double)'
  // note: template argument deduction/substitution failed:
  // note: deduced conflicting types for parameter 'T' ('int' and 'double')
}
```

- *Resolution:*
  - According to the *first function argument* (literal of type `int`), the template argument is deduced as `int`.
  - According to the *second function argument* (literal of type `double`), the template argument is deduced as `double`.
  - None of these options has a higher priority; they are equivalent for the compiler.
  - ⇒ The compiler cannot decide which one to use.

Forwarding reference revisited

- *Example* — *forwarding reference* as a function template parameter:

```
template <typename T> void f(T&& param) { }
```

- It can bound both *lvalue* and *rvalue* arguments of *any type*…
- …and it has special *template argument deduction rules* different for both these cases:
  - In case of *lvalue argument* of type X, T is deduced as X&.
  - In case of *rvalue argument* of type X, T is deduced as X.

```
int i = 1;
f(i);     // i    is lvalue of type int  => T is deduced as int& => type of param is int&
f(true);  // true is rvalue of type bool => T is deduced as bool => type of param is bool&&
```

- Effectively *equivalent source code* after instantiation:

```
void f<int&>(int& param) { }
void f<bool>(bool&& param) { }
```

```
int i = 1;
f(i);
f(true);
```

- *Note* — once instantiation takes place, forwarding reference becomes either *lvalue* or *rvalue reference*.
  - ⇒ There are no more forwarding references after template instantiation.

Template argument substitution

- Once the template argument is resolved (explicitly provided, deduced,…) it is *substituted for the corresponding template parameter*.
- Substitution of template arguments effectively results in an instance of that template.
- *Example:*

```
template <typename T> bool larger(const T& a, const T& b) { return a > b; }

int main() {
  std::cout << larger(1, 2);
}
```

- Here, template argument is deduced as `int` according to function arguments (both of type `int`).
  - ⇒ `int` is substituted for `T`.
  - ⇒ Effectively *equivalent source code* after instantiation:

```
bool larger<int>(const int& a, const int& b) { return a > b; }

int main(int argc, char* argv[]) {
  std::cout << larger<int>(1, 2);
}
```

Template specialization

- *Template specialization* allows having multiple definitions of the same template…
- …while the one used for instantiation is selected according to the template arguments.
- One definition needs to be generic — it is called *primary template*.
- Other definitions are used for instantiation in special cases only ⇒ they are called "*specializations*".
- *Instantiation:*
  - When the template arguments match some specialization, it is used for instantiation.
  - Otherwise, primary template is used instead.
- Typically, only *class templates* are specialized in practice.
  - With *function templates*, *overloading* is preferred.
  - Mixing overloading with function template specialization is "*dangerous*".

Template specialization — *examples*

- *Example* — class that *maps* type to a static Boolean constant which has value:
  - *true* if the type is bool,
  - *false* otherwise.

```
template <typename T>                        // primary template
struct is_bool { static const bool value = false; };

template <>  // specialization for bool template argument
struct is_bool<bool> { static const bool value = true; };
```

```
int main() {
  std::cout << is_bool<int>::value;   // (1) prints out "0"
  std::cout << is_bool<bool>::value;  // (2) prints out "1"
}
```

- *Resolution:*
  - *In (1)*, int *does not* match the specialization ⇒ class is_bool<int> is instantiated from the *primary template*.
  - *In (2)*, bool *does* match the specialization ⇒ class is_bool<bool> is instantiated from this *specialization*.
- ⇒ Effectively equivalent code after instantiation:

```
struct is_bool<int> { static const bool value = false; };
struct is_bool<bool> { static const bool value = true; };

int main() {
  std::cout << is_bool<int>::value;
  std::cout << is_bool<bool>::value;
}
```

Template specialization — examples *(cont.)*

- *Another example* — factorial calculation.
  - *Primary template* is based on recurrent factorial calculation:

```
template <int I>                                                   // primary template
struct factorial { static const int value = I * factorial<I-1>::value; };
```

  - Each recurrence needs some base case = *specialization* in our case:

```
template <> struct factorial<0> { static const int value = 1; };        // specialization
```

```
int main() {
  std::cout << factorial<3>::value;  // prints out 6
}
```

- *Resolution:*
  - Class `factorial<3>` does not match the specialization ⇒ it is instantiated from the *primary template* as:

```
struct factorial<3> { static const int value = 3 * factorial<2>::value; };
```

  - This instance (class) contains the use of template `factorial` with template argument 2 ⇒ it needs to be instantiated as well.
  - Class `factorial<2>` does not match the specialization ⇒ it is instantiated (again) from the *primary template* as:

```
struct factorial<2> { static const int value = 2 * factorial<1>::value; };
```

Template specialization — examples *(cont.)*

```
template <int I>                                                  // primary template
struct factorial { static const int value = I * factorial<I-1>::value; };
template <> struct factorial<0> { static const int value = 1; };  // specialization
```

```
struct factorial<2> { static const int value = 2 * factorial<1>::value; };
```

- *Resolution (cont.):*
  - This instance (class) contains the use of template `factorial` with template argument 1 ⇒ it needs to be instantiated as well.
  - Class `factorial<1>` does not match the specialization ⇒ it is instantiated (again) from the *primary template* as:

```
struct factorial<1> { static const int value = 1 * factorial<0>::value; };
```

  - This instance (class) contains the use of template `factorial` with template argument 0 ⇒ it needs to be instantiated as well.
  - Class `factorial<0>` does match the specialization ⇒ it is instantiated from the specialization as:

```
struct factorial<0> { static const int value = 1; };
```

Template specialization — examples *(cont.)*

- *Summary:*

```
template <int I>                                                      // primary template
struct factorial { static const int value = I * factorial<I-1>::value; };

template <> struct factorial<0> { static const int value = 1; };      // specialization

int main() {
  std::cout << factorial<3>::value;  // prints out 6
}
```

- Effectively equivalent code after instantiation…

```
struct factorial<0> { static const int value = 1; };
struct factorial<1> { static const int value = 1 * factorial<0>::value; };
struct factorial<2> { static const int value = 2 * factorial<1>::value; };
struct factorial<3> { static const int value = 3 * factorial<2>::value; };

int main() {
  std::cout << factorial<3>::value;
}
```

- …and after resolution of the constants:

```
struct factorial<0> { static const int value = 1; };
struct factorial<1> { static const int value = 1; };
struct factorial<2> { static const int value = 2; };
struct factorial<3> { static const int value = 6; };

int main() {
  std::cout << factorial<3>::value;
}
```

Template metafunctions and type traits

- *Recall* — class templates that maps template arguments (types or compile-time constants) to other types or compile-time constants are generally called template metafunctions.
- Both introduced templates `is_bool` and `factorial` fall into this category.
  - `is_bool` maps *type* to *compile-time Boolean constant*,
  - `factorial` maps *compile-time integral constant* to other *compile-time integral constant*.
- Metafunctions that works with types are called "*type traits*".
- We have already seen some type traits from the C++ standard library, such as `std::is_reference` or `std::remove_reference`.
- *Note* — definition of metafunctions is mostly based on template specialization.

Template metafunctions — examples

- Metafunction `std::is_reference` — maps type to a Boolean constant which is:
  - *true* if type is a *reference type*,
  - *false* otherwise.
- *Solution:*
  - *primary template* for non-reference types,
  - *specialization* for reference types.

```
template <typename T> struct is_reference       // primary template
{ static const bool value = false; };

template <typename T> struct is_reference<T&>   // specialization for lvalue references
{ static const bool value = true; };

template <typename T> struct is_reference<T&&>  // specialization for rvalue references
{ static const bool value = true; };
```

- *Partial vs full specialization:*
  - If a specialization is no more parametrized, it is called "*full specialization*".
  - If a specialization is parametrized, it is called "*partial specialization*".

```
template <> struct factorial<0> { static const int value = 1; };  // (full) specialization
```

Template metafunctions — examples *(cont.)*

- *Example:*

```
template <typename T> struct is_reference { static const bool value = false; };
template <typename T> struct is_reference<T&> { static const bool value = true; };
template <typename T> struct is_reference<T&&> { static const bool value = true; };

int main() {
  std::cout << is_reference< float >::value;  // prints out "0"
  std::cout << is_reference< bool& >::value;  // prints out "1"
  std::cout << is_reference< int&& >::value;  // prints out "1"
}
```

- *Resolution:*
  - `float` is *non-reference type* ⇒ it does not match any specialization ⇒ is_reference<float> is instantiated from primary template as:

```
struct is_reference<float> { static const bool value = false; };
```

  - `bool&` is *reference type* ⇒ it does match a specialization for *lvalue references* ⇒ `is_reference<bool&>` is instantiated from it as:

```
struct is_reference<bool&> { static const bool value = true; };
```

  - `int&&` is *reference type* ⇒ it does match a specialization for *rvalue references* ⇒ `is_reference<int&&>` is instantiated from it as:

```
struct is_reference<int&&> { static const bool value = true; };
```

Template metafunctions — examples *(cont.)*

- *Summary:*

```
template <typename T> struct is_reference { static const bool value = false; };
template <typename T> struct is_reference<T&> { static const bool value = true; };
template <typename T> struct is_reference<T&&> { static const bool value = true; };

int main() {
  std::cout << is_reference< float >::value;  // prints out "0"
  std::cout << is_reference< bool& >::value;  // prints out "1"
  std::cout << is_reference< int&& >::value;  // prints out "1"
}
```

- Effectively equivalent code after instantiation:

```
struct is_reference<float> { static const bool value = false; };
struct is_reference<bool&> { static const bool value = true; };
struct is_reference<int&&> { static const bool value = true; };

int main() {
  std::cout << is_reference< float >::value;
  std::cout << is_reference< bool& >::value;
  std::cout << is_reference< int&& >::value;
}
```

Template metafunctions — examples *(cont.)*

- Metafunction `std::remove_reference` — maps input type T to an output type which is:
  - type referred to by T if T is a *reference type*,
  - T otherwise (*identity*).
- *Solution:*
  - *primary template* for non-reference types,
  - *specialization* for reference types.

```
template <typename T> struct remove_reference      { using type = T; }
template <typename T> struct remove_reference<T&>  { using type = T; }
template <typename T> struct remove_reference<T&&> { using type = T; }
```

- *Example:*

```
int main() {
  remove_reference< bool >::type b = true;  // type of b is bool
  remove_reference< int& >::type i = 1;     // type of i is int
}
```

- Effectively equivalent code after instantiation:

```
struct remove_reference<bool> { using type = bool; }
struct remove_reference<int&> { using type = int; }

int main() {
  remove_reference< bool >::type b = true;
  remove_reference< int& >::type i = 1;
}
```

Template metafunctions — “shortcuts”

- *Recall* — usage of metafunctions/type traits requires relatively long writing:

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

- For type traits which “produce” types, we can create a “shortcut” type (alias) template wrapper as follows:

```
template <typename T>
using remove_reference_t = typename remove_reference<T>::type;
```

- Consequently…

```
typename remove_reference<T>::type
```

- …may be then replaced by:

```
remove_reference_t<T>
```

- For type traits in the C++ standard library, such “shortcut” type templates were introduced in C++14.

```
template <typename T> std::remove_reference_t<T> && move(T&& param)
{  return static_cast< std::remove_reference_t<T> && > param;  }
```

Template metafuncts — “shortcuts” *(cont.)*

- Similarly, such “shortcut” wrappers may be created for type traits which “produce” values.
- However, these require variable templates, which were not available until C++17.

```
template <typename T>
inline bool is_reference_v = is_reference<T>::value;
```

- Consequently…

```
is_reference<T>::value
```

- …may be then replaced by:

```
is_reference_v<T>
```

- For type traits in the C++ standard library, such “shortcut” variable templates were introduced in C++17.

```
std::is_reference_v<T>  // since C++17
```

Template specialization — example

- Template specialization has many uses other than just template metafunctions.
- C++ standard library examples:
  - `std::atomic` — specializations for integers, pointers,...;
  - `std::hash` — specialization for different types;
  - `std::vector<bool>` — possible bit-compressed implementation.
- *Another example* — some parallel computation based on MPI (see NI(E)-PDP):
  - MPI library has a "C" API (no templates).
  - MPI functions that send/receive data takes:
    - a *non-type pointer* to these data (of type `void*`),
    - *special argument/"type tag"* that tells the library of what type these data are.
  - *Exemplary MPI type tags:* `MPI_INT` for `int`, `MPI_DOUBLE` for `double`,...
  - Exemplary function for sending data to another MPI process:

```
int MPI_Send(void* buf, int count, MPI_Datatype mdt, int dest, int tag, MPI_Comm comm);
```

Template specialization — example *(cont.)*

- If we know the type of sent data, everything works fine:

```
const int n = 4;
double a[] = { 1.0, 2.0, 3.0, 4.0 };

MPI_Send(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Send(a, n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
```

- *Problem* — what if type of sent data is a template parameter?
- *Example* — parallel computation parametrized by floating-point data type, which defines its used precision:

```
template <typename T>  // T allows to choose FP precision (expected float or double)
void some_parallel_computation() {
  std::vector<T> v;    // result of local process
  ...                  // calculate local result
  // send it to root process (process number 0):
  MPI_Send(v.data(), v.size(), MPI_???, 0, 0, MPI_COMM_WORLD);
```

- How to specify *MPI type tag* that specifies type of sent data?
- Required functionality:
  - If T is `float`, `MPI_FLOAT` type tag should be used.
  - If T is `double`, `MPI_DOUBLE` type tag should be used.
- Generally, we need to map types to corresponding *MPI type tags*.

Template specialization — example *(cont.)*

- *Possible solution* — template specialization.
  - *Note* — *MPI type tags* may be in fact pointers into some type table (link).
  - ⇒ Generally, they may not be compile-time constants.
  - *Alternative* — static member function that returns tag value.

```
template <typename T> struct GetMPIDatatype;  // primary template; declaration only
template<> struct GetMPIDatatype<float>
{  static MPI_Datatype get() { return MPI_FLOAT; }  };   // specialization for float
template<> struct GetMPIDatatype<double>
{  static MPI_Datatype get() { return MPI_DOUBLE; }  };  // specialization for double
... // similarly for other types supported by MPI library (MPI_INT, MPI_LONG,...)
```

- Application to our problem:

```
MPI_Send(v.data(), v.size(), GetMPIDatatype<T>::get(), 0, 0, MPI_COMM_WORLD);
```

- *Example* — computation to be instantiated with *single precision*:

```
MPI_Send(v.data(), v.size(), GetMPIDatatype<float>::get(), 0, 0, MPI_COMM_WORLD);
```

- This initiates instantiation of `GetMPIDatatype<float>` from a corresponding specialization:

```
struct GetMPIDatatype<float>
{  static MPI_Datatype get() { return MPI_FLOAT; }  };
```

Templates — complex example

- Consider our implementation of Vector class template...

```
template <typename T> class Vector {
  size_t capacity_, size_;
  T* data_;
  ...
```

- ...and its following exemplary use:

```
struct X {
  X(int);         // converting constructor
  X(int, bool);   // another converting constructor
  X(const X&);    // copy constructor
  X(X&&);         // move constructor
};

int main() {
  Vector<X> v;
}
```

- Vector class template is instantiated with X template argument, which results in Vector<X> class ⇒ X is substituted for T:

```
class Vector<X> {
  size_t capacity_, size_;
  X* data_;
  ...
```

- Let us now look at some Vector member functions.

Templates — complex example *(cont.)*

- First, the reserve member function was defined as:

```
void reserve(size_t capacity) {
  if (capacity <= capacity_) return;
  T* data = (T*)::operator new(capacity * sizeof(T));

  for (size_t i = 0; i < size_; i++) new (data + i) T( move( *(data_ + i) ) );

  clear();
  ::operator delete(data_);
  data_ = data; capacity_ = capacity;
}
```

- After *instantiation/substitution*, this effectively results in the following definition:

```
void reserve(size_t capacity) {
  if (capacity <= capacity_) return;
  X* data = (X*)::operator new(capacity * sizeof(X));

  for (size_t i = 0; i < size_; i++) new (data + i) X( move( *(data_ + i) ) );

  clear();
  ::operator delete(data_);
  data_ = data; capacity_ = capacity;
}
```

- This initiates the instantiation of move function template.
- *Note* — we deliberately used our custom version here instead of `std::move` to show what happens next.

Templates — complex example *(cont.)*

- Our custom move function template definition:

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

- *Note* — similarly as above, we used our remove_reference trait.
- *Resolution:*
  - *Template argument* of move call is not specified.
    - ⇒ It needs to be deduced according to the *function argument*.
  - *Function argument* of move is *lvalue* expression of type X.
  - param is a *forwarding reference*.
    - ⇒ T is deduced as X&.
  - This effectively results in the following instantiation of move:

```
remove_reference<X&>::type && move<X&>(X& && param) {
  return static_cast< remove_reference<X&>::type && > param;
}
```

- Now, this instantiation initiates instantiation of remove_reference<X&> class.

Templates — complex example *(cont.)*

- `remove_reference<X&>` is instantiated as follows:

```
struct remove_reference<X&> { using type = X; }
```

- ⇒ *For illustration*, we can then effectively rewrite…

```
remove_reference<X&>::type && move<X&>(X& && param) {
  return static_cast< remove_reference<X&>::type && > param;
}
```

- …as…

```
X&& move<X&>(X& param) {  return static_cast< X&& > param;  }
```

- …and this instance of move is called in `Vector<X>::reserve`:

```
void reserve(size_t capacity) {
  ...
  for (size_t i = 0; i < size_; i++) new (data + i) X( move<X&>( *(data_ + i) ) );
  ...
```

- This is exactly what we expected — `move<X&>` call is an expression that:
  - represents the original vector element `*(data_ + i)` of type X,
  - and its category is *rvalue*.
- ⇒ New elements are initialized by *move constructor* of class X.

Templates — complex example *(cont.)*

- *Summary* for `Vector<X>::reserve`…

```
template <typename T> class Vector {
  size_t capacity_, size_;
  T* data_;
public:
  void reserve(size_t capacity) {
    if (capacity <= capacity_) return;
    T* data = (T*)::operator new(capacity * sizeof(T));
    for (size_t i = 0; i < size_; i++) new (data + i) T( move( *(data_ + i) ) );
    clear();
    ::operator delete(data_);
    data_ = data; capacity_ = capacity;
  }
```

- …is effectively instantiated as follows:

```
struct remove_reference<X&> { using type = X; }
remove_reference<X&>::type && move<X&>(X& param) {
  return static_cast< remove_reference<X&>::type && > param;
}
class Vector<X> {
  size_t capacity_, size_;
  X* data_;
public:
  void reserve(size_t capacity) {
    if (capacity <= capacity_) return;
    X* data = (X*)::operator new(capacity * sizeof(X));
    for (size_t i = 0; i < size_; i++) new (data + i) X( move<X&>( *(data_ + i) ) );
    clear();
    ::operator delete(data_);
    data_ = data; capacity_ = capacity;
  }
```

Templates — complex example *(cont.)*

- Next, the push_back member functions were defined as:

```
void push_back(const T& param) {                                 // overload for lvalues
  if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_) T( param );
  size_++;
}

void push_back(T&& param) {                                      // overload for rvalues
  if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_) T( move( param ) );
  size_++;
}
```

- After *instantiation/substitution*, this effectively results in the following definitions:

```
void push_back(const X& param) {
  if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_) X( param );
  size_++;
}

void push_back(X&& param) {
  if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_) X( move<X&>( param ) );
  size_++;
}
```

- *Note* — move call after deduction results in the call of the already instantiated function move<X&>.

Variadic templates

- What about `emplace_back` member function?

```
template <typename... U> void emplace_back(U&&... param) {
  if (size_ == capacity_) reserve(capacity_ ? 2 * capacity_ : 1);
  new (data_ + size_) T( std::forward<U>(param)... );
  size_++;
}
```

- To understand it, we first need to explain *variadic templates*.
- *Recall* — *template parameter* starts with
  - `typename` (or `class`) keyword for *type template parameters*,
  - *type name* for *non-type template parameters*,
- and is then it is (optionally) followed by identifier (template parameter name).

```
template <typename T, class U, int I, size_t N> struct X { };
```

- When we write *ellipsis* (sequence of 3 dots) after the typename/class/*type name*, we create a so-called "*template parameter pack*" (instead of a *template parameter*).

```
template <typename T, typename ... Ts> struct X { };
  // T is template parameter, Ts is a template parameter pack
```

- Template with at least one template parameter pack is called **variadic template**.

Variadic templates *(cont.)*

- *Recall* — when template is instantiated, *template parameters* are substituted by *template arguments*.
- Namely, a single template parameter is substituted by a single template argument.
- On the contrary, *template parameter pack* may be substituted by an arbitrary number of template arguments, including no one.

```
template <typename T, typename ... Ts> struct X { };

X<int, bool, double> x1;  // T is substituted by int, Ts is substituted by bool and double
X<int, bool> x2;          // T is substituted by int, Ts is substituted by bool
X<int> x3;                // T is substituted by int, Ts is empty
```

- *Effectively*, template parameter pack represents a list of virtual *unnamed* template parameters.
- It is not possible to refer to individual elements of template parameter pack.
- However, it is possible to so-call "*expand*" it.

Variadic templates — expansion

- *Template parameter pack expansion* happens when its *identifier* is followed by *ellipsis* in the template definition.
- *Effect* — as if the list of virtual template parameters were written in the source code separated by commas.
- Within *instantiation*, this list is substituted by the list of comma-separated template arguments.

```
template <typename T, typename ... Ts> struct X { Ts... };
X<int, bool, double> x;    // Ts is substituted by bool and double
                           // => Ts... is literaly expanded to "bool, double"
```

- ⇒ Instantiated class `X<int,bool,double>`:

```
struct X<int, bool, double> { bool, double };  // for illustration, does not compile
```

- Where such expansion can be used?
  - Wherever a *comma-separated list of template parameters/arguments* can be used.
  - In case of type template parameter list ⇒ wherever one can use a *comma-separated list of types*.

Variadic templates — tuple

- For instance, a comma-separated list of types can be used as a list of template arguments.
- *Example — tuple-like class*.
- *Tuple* = generalized pair.
- `std::pair` is a class template that accepts two type template arguments.
- `std::pair<T1,T2>` class is an "owner" that owns/manages two objects.
  - First object is of type `T1`, second object is of type `T2`.
  - Both objects are stored in the "included" storage of the pair owner.
- `std::tuple` class template generalizes this concept to any number of owned/managed objects.
  - ⇒ An arbitrary number of type template arguments needs to be accepted.
  - ⇒ `std::tuple` is a variadic template.

std::tuple

Defined in header `<tuple>`

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

Variadic templates — tuple *(cont.)*

- How to define a tuple-like class template?

```
template <typename... Ts>
class Tuple {  /* ??? */ };
```

- We would want to define member variables — one for each template argument (having its type).
- Unfortunately, this is not directly possible with the template parameter pack expansion mechanism.
- *Possible solution* — recurrent definition:
  - Tuple with some template arguments can be defined as a class with the following member variables:
    - First member variable has the type of the first template argument.
    - Second member variable is a tuple with the remaining template arguments.
  - ⇒ We need to "*recognize*" the first template argument.
  - ⇒ It cannot be "*mapped*" to the template parameter pack.

```
template <typename T, typename... Ts>
class Tuple {
  T t_;
  Tuple< Ts... > ts_;
  // ... other code (constructors, member functions, etc.)
```

Variadic templates — tuple *(cont.)*

- As always with recurrence, we need some "*base case*".
- In our case, it will be a *specialization for a single template argument*:

```
template <typename T, typename... Ts>  // primary template
class Tuple {
  T t_;
  Tuple< Ts... > ts_;
  // no other code now
};

template <typename T>                  // specialization for a single template argument
class Tuple<T> {
  T t_;
};
```

- *Note* — instantiation of `Tuple` template with a single template argument in fact corresponds with both the primary template (with empty parameter pack) and the specialization.
- However, C++ standard rules gives higher "priority" to the specialization.

```
Tuple<int> t;  // no ambiguity; will be instantiated from the specialization
```

Variadic templates — tuple *(cont.)*

- *Example:*

```
template <typename T, typename... Ts> class Tuple {
  T t_;
  Tuple< Ts... > ts_;
};

Tuple< int, bool, double > t;  // T substituted for int, Ts for bool and double
                               // Ts... is expanded to bool, double
```

- It does not match the specialization ⇒ it will be instantiated from the primary template as:

```
class Tuple< int, bool, double > {
  int t_;
  Tuple< bool, double > ts_;
};
```

- This instantiation initiates the instantiation of `Tuple` with `bool` and `double` template arguments.
- This is, again, instantiated from the primary template as:

```
class Tuple< bool, double > {
  bool t_;
  Tuple< double > ts_;
};
```

Variadic templates — tuple *(cont.)*

```
class Tuple< bool, double > {
  bool t_;
  Tuple< double > ts_;
};
```

- This instantiation initiates the instantiation of `Tuple` with `double` template argument.
- This instantiation matches both primary template and specialization, but specialization gets priority.

```
class Tuple< double > {
  double t_;
};
```

- *Summary* of instantiated `Tuple` classes:

```
class Tuple< double > {
  double t_;
};

class Tuple< bool, double > {
  bool t_;
  Tuple< double > ts_;
};

class Tuple< int, bool, double > {
  int t_;
  Tuple< bool, double > ts_;
};
```

Variadic templates — tuple *(cont.)*

- The initial instantiation is kind-of effectively equivalent to the following one:

```
class Tuple< int, bool, double > {
  int t_;

  class Tuple< bool, double > {
    bool t_;

    class Tuple< double > {
      double t_;
    } ts_;

  } ts_;
};
```

- ⇒ The instance effectively — but not directly — contains member variables of types of all template arguments.
- *Notes:*
  - This is a basic definition useless for practice.
  - However, it showed how variadic templates and templates parameter packs are commonly defined and resolved in practice.
  - For practical purposes, we would need to add constructors, functions for accessing owned objects, etc., which is beyond the scope of this lecture.