NICOLAI M. JOSUTTIS
solutions in time

C++ Templates, 2nd Ed.: Errata for all Printings

C++ Templates - The Complete Guide, Second Edition

Errata for all Printings

June 26, 2019

This is the errata of the 1st printing of the book ***C++ Templates, 2nd Edition*** by David Vandevoorde, Nicolai M. Josuttis, and Douglas Gregor.
It covers all errors that were found.
The errata is organized in the following way:
- The first part lists technical errors
- The second part lists typos

Errors

In general, throughout the book: 2017-09-01

We forgot to make it more clear that before C++17 you have to add a definition to a static constexpr declaration of a class member.

```
struct A {
  static constexpr int n = 5; // C++11/C++14: declaration, since C++17: definition
};
```

Before C++17, this is a declaration of n only. Only since C++17 is this also a definition. So, before C++17 in exactly one translation unit you have to provide a corresponding definition, which we also missed in several examples:

```
// in one translation unit before C++17:
constexpr int A::n; // C++11/C++14: definition (since C++17: deprecated redundant declaration)
```

The reason, nobody saw this before the book was out is that often the definition is not necessary at all. In fact, you can always pass the member by value. Only when passing these members by reference and if the compiler doesn't optimize the call away link errors occur:

```
std::cout << A::n;          // OK even without definition (ostream::operator<<(int) gets A::n by value)

int inc(const int& i);      // pass by reference!
std::cout << inc(A::n);     // link error without definition and strong optimizations
```

2.8, page 39, Alias Templates for Member Types: 2019-06-26

There are a couple of issue with the whole subsection:

```
s/struct C {/template<typename T> struct MyType {/

s/struct MyType{/template<typename T> struct MyType {/
```

s/allows the use of**/**allows the generic use of**/**

s/`MyTypeIterator<int> pos;` / `MyTypeIterator<T> pos;` /

3.1, page 46, basics/stacknontype.hpp: 2017-10-13

twice: s/assert(!elems.empty());**/**assert(!empty());**/**

3.3, page 49: 2017-09-20

The requirements for linkage in the first bullet list use the wrong C++ versions. The correct bullets are:

- **In C++98 and C++03**, the objects also had to have external linkage.
- **In C++11 and C++14**, the objects also had to have external or internal linkage.

(the comments in the example code below shows it correctly) **and right behind it:**
s/`class MyClass {`**/**`class Message {  // OK` **/**
and
s/`MyClass<"hello"> x;  // ERROR: string literal "hello" not allowed`**/**`Message<"hello"> x;  // ERROR: string literal "hello" not allowed`**/**

3.4, page 51, basics/stackauto.hpp: 2017-10-13

twice: s/assert(!elems.empty());**/**assert(!empty());**/**

15.5, page 78: 2019-01-13

s/Member function templates can also be partially or fully specialized.**/**Member function templates can also be fully specialized, but not partially specialized.**/**

13.1, page 216, table 13.1. 2017-10-24

The footnote text for Operator-function-id is missing. It should be:

> Many operators have alternative representations. For example, `operator &` can equivalently be written as `operator bitand` even when it denotes the unary *address of* operator.

16.3.2, page 342, on Full Function Template Specialization:2017-10-27

We didn't make it clear that a full function template specialiation only provides an alternative *definition* but not an alternative *declaration*. For this reason, the signature (including the return type) must match exactly:

```
template<typename T> auto foo();
template<> int foo<int>() { return 42; }  // ERROR
template<> auto foo<int>() { return 42; } // OK
```

Typos

Page XXV, Achknowledgements for 2nd Ed.: s/Nevin Lieber**/**Nevin Liber**/**

Page 7, 1.1.3, Footnote: s/Visual Studio 20133**/**Visual Studio 2013**/**

Page 8, 1.2, Type Conversions During Type Deduction: before: `int const c = 42;` is missing: `int i = 17;`
and: s/`foo(&i, arr);`**/**`max(&i, arr);`**/**

and later: s/`foo("hello", s); // ERROR: T can be deduced as char const[6] or std::string`**/**`max("hello", s); // ERROR: T can be deduced as char const* or std::string`**/**

Page 12, 2nd but last para of 1.3.2: s/by the standard library in `<type_trait>` (see Section D.5 on page 732).**/**by the standard library in `<type_traits>` (see Section D.4 on page 731).**/**

Page 14, 1.4: s/to depend from previous template parameters**/**to depend on previous template parameters**/**

Page 18, 1.5, basics/max3ref.cpp: s/ // run-time ERROR **/** // run-time ERROR (undefined behavior) **/**

Page 24, 2.1, basics/stack1.hpp top(): s/ // return copy of last element **/** // return ~~copy of~~ last element **/**

Page 26, 2.1.2: s/This is also done in `top()`, ... on attempts to remove a nonexistent top element:**/**This is also done in `top()`, ... on attempts to access a nonexistent top element:**/**

Page 27, 2.1.2, top: s/ // return copy of last element **/** // return ~~copy of~~ last element **/**

Page 29, 2.3: s/`void printOn() (std::ostream& strm) const {`**/**`void printOn(std::ostream& strm) const {`**/**

Page 30, 2.4: s/`void printOn() (std::ostream& strm) const {`**/**`void printOn(std::ostream& strm) const {`**/**

Page 33, 2.5, basics/stack2.hpp: s/ // return copy of last element **/** // return ~~copy of~~ last element **/**

Page 36, 2.7, basics/stack3.hpp top(): s/ // return copy of last element **/** // return ~~copy of~~ last element **/**

Page 38, 2.8, footnote: s/ in intentional **/** is intentional **/**

Page 39, 2.8, Alias Templates for Member Types: s/`struct C {`**/**`template <typename T> struct MyType {`**/**
and: s/`struct MyType {`**/**`template <typename T> struct MyType {`**/**

Page 40, 2.9, 1st para: s/ (that don't have a default value), **/** (that don't have a default value)**.** **/**
and: s/ Instead, you can skip to define the templates arguments explicitly, **/** Instead, you can skip defining the templates arguments explicitly, **/**

Page 40, 2.9, 1st code snippet: s/`Stack<int> intStack1; // stack of strings`**/**`Stack<int> intStack1; // stack of ints`**/**

Page 41, 2.9, 1st bullet: s/Due to the definition of the int constructor, **/**Due to the definition of the constructor template, **/**

Page 41, 2.9, header: s/ Class Template Arguments Deduction with String Literals / Class Template Argument Deduction with String Literals /

Page 47, 3.1: s/The new second template parameter, `Maxsize`, is of type `int`.**/**The new second template parameter, `Maxsize`, is of type `std::size_t`.**/**

Page 48, 3.2: s/and `std::transform()` need a complete type**/**and `std::transform()` need**s** a complete type**/**
and: s/the see, what could fit,**/**the**n** see, what could fit,**/**

Page 50, 3.3: s/In all three cases a constant character array is initialized by "hello"**/**In all three cases a constant character array is initialized by **"hi"/**

Page 56, 4.1.1: s/Thus, after printing `"world"` as `firstArg`, we calls `print()` with no arguments**/**Thus, after printing `"world"` as `firstArg`, we call `print()` with no arguments**/**

Page 61, 4.3, first code snippet: s/`v.emplace("Tim", "Jovi", 1962);`**/**`v.emplace_back("Tim", "Jovi", 1962);`**/**

Page 62, 4.4.1, mid code snippet: s/`std::complex<float>(4,2) + std::complex<float>(4,2);`**/**`std::complex<float>(4,2) + std::complex<float>(4,2));`**/**

Page 69, 5.2: twice as code comment: s/ // x is zero (or false) if `T` is a built-in type**/** // x is zero (or `false` or `nullptr`) if `T` is a built-in type**/**

Page 87, 5.7, basics/stack9.hpp top(): s/ // return copy of last element **/** // return ~~copy of~~ last element **/**

Page 93, 6.1, first paragraph: s/instead of `g(&&)` would be called.**/**instead of `g(X&&)` would be called.**/**

Page 93, 6.1, first code snippet: s/`void f(T val) {`**/**`void f(T& val) {`**/** **and** **s/**`g(T);`**/**`g(val);`**/**

Page 101, 6.4, bottom: s/However, the order of the arguments is the opposite is this case:**/**However, the order of the arguments is the opposite **in** this case:**/**

Page 108, 7.1, code in "Passing by Value Decays": s/`printV(arr);` // decays to pointer so that arg has type `char const*`**/**`printV(arr);` // decays to pointer so that arg has type `int*`**/**

Page 112, 7.2.3, last line of last but one code example: s/`passR("hi");` // OK: T deduced as `int (&)[4]` (also the type of `arg`) **/** `passR(arr);` // OK: T deduced as `int (&)[4]` (also the type of `arg`)**/**

Page 112, 7.2.3: last code example: s/`foo(42);`**/**`passR(42);`**/** **and** **s/**`foo(i);`**/**`passR(i);`**/**

Page 113 top, 7.3: code example: s/`printT(s);` // pass s by reference**/**`printT(s);` // pass s by value**/**

Page 114, 7.3: s/if you pass objects through generic code**/**if you *pass* objects through generic code to nongeneric functions**/**

Page 114, 7.3: s/ `if (isless(std::cref(s) < "world"))` ... // ERROR **/** `if (isless(std::cref(s), "world"))` ... // ERROR **/**
and: s/ `if (isless(std::cref(s) < std::string("world")))` ... // ERROR **/** `if (isless(std::cref(s), std::string("world")))` ... // ERROR **/**

Page 131, 8.4: s/(match with ellipsis (...) in overload resolution (see Section C.2 on page 682).**/**(match with ellipsis (...) in overload resolution**,** see Section C.2 on page 682).**/**

Page 132; 8.4, real std::thread definition: s/typename = std::enable_if_t<!std::is_same_v<std::decay_t<F>,**/** typename = enable_if_t<!is_same_v<decay_t<F>,**/**

Page 133, 8.4.1: s/decltype((void)(t.size)(), T::size_type())**/**decltype((void)(t.size()), T::size_type())**/**

Page 134, 8.5: s/that allows is to enable or disable**/**that allows us to enable or disable**/**

Page 137, 9.1.1, first bullet: s/Classes and other types are entirely placed in header files.**/**Classes and other type declarations are entirely placed in header files.**/**

Page 147, 9.4: line numbering wrong after "immediately followed by more than 20 other error messages:**": First line be 17 instead of 16 (or all following one less)**

Page 165, 11.2.1: s/`is_swappable_v<int&,int&>` // yields true (equivalent to the previous check)**/**`is_swappable_with_v<int&,int&>` // yields true (equivalent to the previous check)**/**

Page 169, 11.4, basics/referror2.cpp: s/loops over 120 elements**/**loops over 110 elements**/**

Page 181, 12.1, Nontemplate Members of Class Templates, 1st para: s/a class template They are occasionally**/**a class template**.** They are occasionally**/**

Page 204, 12.4.2: s/`Values` is a nontype template parameter pack...**/**`Vs` is a nontype template parameter pack...**/**
Page 204, 12.4.2: s/...provided for the template type parameter type `Types`.**/**...provided for the template type parameter type `Ts`.**/**
Page 204, 12.4.2: s/Note that the ellipsis in the declaration of `Values`...**/**Note that the ellipsis in the declaration of `Vs`...**/**

Page 204, 12.4.3, after code with main(): s/type of the i^{th} value in `Values`**/**type of the i^{th} value in `values`**/**

Page 206, 12.4.4, last code block: should only have the 01 and 02 expansion (skip comma, `03(...)`**)**

Pagw 208, 12.4.6: s/With this feature available, code like**/**With this feature available, code calling a trait for each passed type T like**/**

Page 209, 12.4.6, close to the end, first bullet: s/the value true,.**/**the value true**.**/**

Page 246, 14.2.2: s/(i.e., the member are partially instantiated).**/**(i.e., the members are partially instantiated).**/**

Page 250, 14.3.1, code snippet: s/`template<typename> void f() {}`**/**`template<typename> void f(T p) {}`**/**
Page 250, 14.3.1, code snippet: s/`template<typename T> void h(T P) {`**/**`template<typename T> void h(T p) {`**/**

Page 262: 14.5.1, second example: s/ // ===== t.hpp: **/** // ===== f.tpp: **/**

Page 278, 15.6.1: s/ `R const& rr = r;` // OK: rr has type int& **/** `RI const& rr = r;` // OK: rr has type int& **/**

Page 283: 15.6.4: s/`template<typename Other> X(X<U>&&);` // X<U> is not a template parameter**/**`template<typename U> X(X<U>&&);` // X<U> is not a template parameter**/**

Page 287: 15.8.1: s/`int r = f(v, 7.0);` // OK: T is deduced to `int` through the first parameter,**/**`int r = f(v, 7.0);` // OK: T is deduced to V through the first parameter,**/**

Page 294, 15.10.1: s/`typename Container::const_iterator pos = container.begin();`**/**`typename Container::iterator pos = container.begin();`**/**

Page 301, 15.10.3 code comment: s/x1 has type int**/**x has type int**/**

Page 316, 15.12.2 code example: s/`S y{s1};`**/**`S y{x};`**/ and s/**`S z(s1);`**/**`S z(x);`**/ and s/**but what should the type of x and y be?**/**but what should the type of y and z be?**/**

Page 326, 16.1.2: s/The prefix `quick_` is helpful**/**The prefix `quick` is helpful**/**

Page 350, 16.4: s/ `impl.append((void* C::*)pm);` **/** `impl.append(static_cast<void* C::*>(pm));` **/**

Page 408, after traits/passbyvalue.cpp: s/described above First, we define**/**described above**.** First, we define**/**

Page 412, traits/issame.cpp: s/ `foo(7.7); // calls fooImpl(42, FalseType)` **/** `foo(7.7); // calls fooImpl(7.7, FalseType)` **/**

Page 417, after traits/isdefaultconstructible1.hpp: s/ `= IsSameT<decltype(test<...>(nullptr)), char>::value;` **/** `= IsSameT<decltype(test<T>(nullptr)), char>::value;` **/**

Page 420, 19.4.2, 1st para after traits/isdefaultconstructible3.hpp: s/because by default a type doesn't have the member size_type.**/**assuming by default a type doesn't have a default constructor.**/**

Page 428, 19.5, traits/isconvertibale.hpp: s/template<typename F, typename T,**/**template<typename F, typename,**/**

Page 430, 19.5: s/but for two `void` types it will produce `false_type`.**/**but for two `void` types it will produce `true_type`.**/**

Page 445, 19.7.2: s/instantiated instead (producing a `std::false_type` result.**/**instantiated instead (producing a `std::false_type` result).**/**

Page 469, 20.3: s/(i.e., it is associated with a tag convertible to `std::random_access_iterator_tag`.**/**(i.e., it is associated with a tag convertible to `std::random_access_iterator_tag`).**/**

Page 476, 20.3.4 s/For example, we expect our overloaded `container` constructors**/**For example, we expect our overloaded `Container` constructors**/**

Page 484, typeoverload/min.cpp: s/ `min(X4(), X4()); // ERROR: X4 can be passed to min()` **/** `min(X4(), X4()); // ERROR: X4 cannot be passed to min()` **/**

Page 494, 21.1.2, inherit/basememberpair.hpp: s/ // access base class data via `first()` **/** // access base class data via `base()` **/**

and: s/ // access member data via `second()` **/** // access member data via `member()` **/**

Page 522, 22.2, bridge/functionptr.hpp, `operator bool()`: s/`return bridge == nullptr;`/`return bridge != nullptr;`/

Page 523, 22.3, top paragraph: s/and an `invoke`. operation to call**/**and an `invoke()` operation to call**/**

Page 666, A.3.1: the second: s/ //====== translation unit 1: **/** //====== translation unit 2: **/**

Home of the C++ Templates book