Department I - C Plus Plus

Modern and Lucid C++ Advanced for Professional Programmers

Week 2 – New Features in C++17

Prof. Peter Sommerlad / Thomas Corbat
Rapperswil, 28.02.2019
FS2019

Cevelop
Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS
INSTITUTE FOR
SOFTWARE

C++17 Features

Cevelop

Your C++ deserves it

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

static_assert Without Message

- **Before C++17: `static_assert` required an expression evaluating to `bool` and a message, to be displayed when the assert failed.**

```
#include <type_traits>

template<typename T>
T negate(T t) {
  static_assert(std::is_signed_v<T>, "negate can only be called on signed types");
  return -t;
}
```

- **Since C++17: The message is optional. The whole `static_assert` will be displayed anyway.**

```
#include <type_traits>

template<typename T>
T negate(T t) {
  static_assert(std::is_signed_v<T>);
  return -t;
}
```

New Initialization Rules for auto with {}

- **The auto keyword has been introduced in C++11. The specification had some counter-intuitive effect:**

```
int int_value = 1;

int int_value{1};
```

⇔ ?

```
auto auto_value = 1;

auto auto_value{1};
```

- **Until C++17**

```
auto auto_value{1};
```

⇔ ?

```
std::initializer_list<int> auto_value{1};
```

- **Already implemented by several compilers for C++11/14**
  - Example with older compiler: https://godbolt.org/g/rDJu6s

Many New Features in the Standard Library

- **`<numerics>`**
  - `gcd` (Greatest Common Divisor)
  - `lcm` (Least Common Multiple)
  - ...
- **`<algorithm>`**
  - `clamp`
  - `reduce`
  - Execution Policies
  - ...
- **`<cstddef>`**
  - `byte` (Byte Type)
- **`<any>`**
- **`<optional>`**
- **`<variant>`**
- **`<filesystem>`**
- **`<string_view>`**
- **...**

std::any

#include <any>

- **Container for single object/value of any type**
- **Can be empty**
  - Check with `has_value()` member function
- **Allows type-safe access to the element**
  - `std::any_cast`
  - throws `std::bad_any_cast` on type mismatch
  - You need to know what you put into the any
- **Application: Replacement of void ***
  - Avoids memory leaks
- **Is part of the `<utility>` header**

```cpp
void any_example(std::ostream & out) {
  std::any value{};
  out << "has value? " << value.has_value() << '\n';
  value = 5;
  out << std::any_cast<int>(value) << '\n';
  try {
    std::any_cast<long>(value);
  } catch(std::bad_any_cast const &) {
    out << "std::bad_any_cast thrown, "
           "when accessing int as long!\n";
  }
}
```

std::variant<T...> (1/2)

`#include <variant>`

- **Container for single object/value of type from given list**
- **Cannot be empty**
  - If empty value is required add the type `std::monostate`
- **Retrieving element**
  - `std::get` or `std::get_if`
  - Throws `std::bad_variant_access` on type mismatch
- **Replacement for union**

```
void variant_example(std::ostream & out) {
  std::variant<int, float, std::string> value{};
  out << std::get<int>(value) << '\n';

  value = "char const [15]";
  try {
    out << std::get<int>(value) << '\n';
  } catch(std::bad_variant_access const &) {
    out << "std::bad_any_cast thrown, "
    "when accessing string as int!\n";
  }

  value = 10L; //Compile error
}
```

std::variant<T...> (2/2)

#include <variant>

- **Visiting element (`std::visit`)**
  - Requires object of type that features an overloaded call operator for every possible element type
  - Overload for active type will be called
- **Two variants of the same type list can be compared**
  - Comparison on the element if the elements have the same active type
  - If type arguments don't have the same order, the variant types are different!

```
void variant_compare() {
  std::variant<int, float> vIF{}, vIF_too{};
  std::variant<float, int> vFI{};
  vIF == vIF_too;
  vIF == vFI; //Compile error
}
```

```
struct VariantHandler {
  std::ostream & out;
  void operator()(int & i) const {
    out << "int: " << i << '\n';
  }
  void operator()(float & f) const {
    out << "float: " << f << '\n';
  }
  void operator()(std::string & s) const {
    out << "string: " << s << '\n';
  }
};

void variant_example(std::ostream & out) {
 //...
  value = 15.0f;
  std::visit(VariantHandler{out}, value);
  //...
}
```

std::optional<T>

`#include <optional>`

- **Many libraries use pointers to represent a potentially absent or erroneous value**
  - Requires heap construction of existing value
    - Can use polymorphism
  - Caller needs to take ownership responsibility, unless `unique_ptr`/`shared_ptr` is used
    - overhead with `shared_ptr`

```
valuetype * doit();

auto result = doit();
if (result) ...
```

- **`std::optional<T>` to the rescue**
  - Return by value (no heap memory)
  - `optional<T>` contains a `T` by value or is empty
- **Was available in `<boost/optional.hpp>` as `boost::optional<T>`**

```
unique_ptr<valuetype> doit();
optional<valuetype> doit();

auto result = doit();
if (result) {
  result->something();
}
```

`std::optional` Example

- **`openFile` shouldn not return an `ifstream` object, if the file does not exist.**
  - one could also return an `eof()` `ifstream`, but…
- **If the file could be opened it should also be usable**
  - `!!optional`
    - Converts to `bool`, `true` when OK
- **Test case cleans up written file using functionality from filesystem TS (next slides)**

```
optional<std::ifstream> openFile(std::string const name) {
  std::ifstream file{name};
  if (file.is_open()) return std::move(file);
  return{};
}

void testNonExistingFile() {
  auto nofile = openFile("gugus.txt");
  ASSERTM("file shouldn't be opened", !nofile);
}

void testExistingFile(){
  std::string const name{"hello.txt"};
  std::string const writtencontent{"hello, world"};
  std::ofstream of{name};
  of << writtencontent <<'\n';
  of.close();
  {
    auto somefile = openFile(name);
    ASSERTM("file should be opened", !!somefile);
    std::string content{};
    std::getline(*somefile, content);
    ASSERT_EQUAL(writtencontent, content);
  }
  fs::remove(name);
}
```

Filesystem Library

`#include <filesystem>`

- **Key abstractions: path, directory-iterators, stati and permission of files/directories**
- **Functions:**
  - “massage” paths (relative, absolute, canonical)
  - Copy, remove and link files
  - Create directories and links
  - Obtain filesystem meta information: sizes, free space, access time and rights, files status

Filesystem Example: Directory Listing

- **directory_iterator is easy to use**
- **recursive_directory_iterator recurses**
- **file_size obtains sizes**
- **is_directory etc. for file type query**

- **Note: you need to add the library “stdc++fs” to your linker command for gcc**
  - Should also work with MinGW-w64 (7.2.0)
- **If your compiler does not support it you can use boost/filesystem instead**

```
#include <filesystem>
namespace fs = std::filesystem;

int main(int argc, char **argv) {
 using std::cout;
 fs::path dir{ "./" };
 if (argc > 1)
  dir = argv[1];
 for (auto p : fs::directory_iterator(dir)) {
  cout << p << '\t';
  if (is_directory(p)) {
   cout << "DIR";
  } else
   try { //no file sizes for directory
    auto sz = fs::file_size(p);
    if (sz > 1024)
     cout << sz / 1024 << " kB";
    else
     cout << sz << " Bytes";
   } catch (...) {
    cout << "no file size available";
   }
  cout << '\n';
 }
}
```

Path

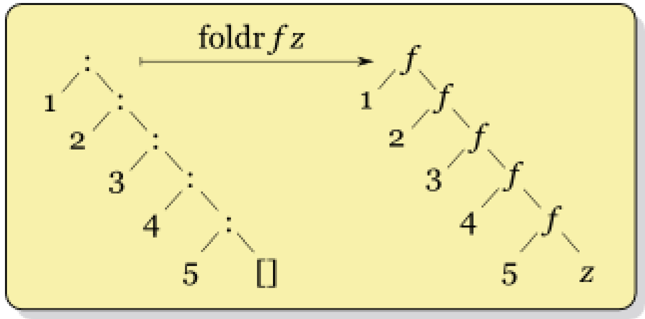
- **“hidden” benefit: unicode string class**
  - Can construct from any kind of string literal or string type `L""`, `u8""`
- **`fs::path` can be constructed from non-ASCII strings**
  - `u8char_t` -> UTF-8 encoded unicode
  - `wchar_t` -> (Windows) wide characters
  - `char` -> ASCII (or locale-based string)
- **It is not always guaranteed, if such a string can be represented in the filesystem underneath**
  - As with all I/O you should expect failures
- **Can use operator / for concatenating paths.**

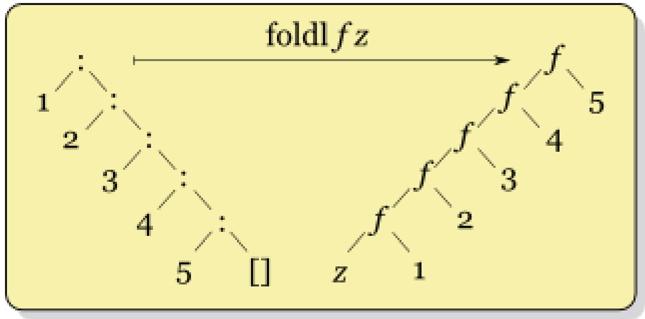
```
#include <filesystem>
namespace fs = std::filesystem;

int main() {
  using std::cout;
  fs::path const p {"./Hallo"};
  cout << p << '\n';
  if (is_directory(p)) {
    cout << " DIR exists" << '\n';
    fs::remove(p);
  } else {
    cout << "creating dir " << p ;
    if (fs::create_directory(p))
      cout << " success";
    else
      cout << " failed";
    cout << '\n';
  }
}
```

Fold Expressions

- **Simplified syntax for reducing variadic template parameter packs**
- **Four forms of fold expressions in variadic templates**
  - `( <pack-name> <op> ... )` – fold right
  - `( ... <op> <pack-name> )` – fold left
  - `( <pack-name> <op> ... <op> <init> )` – fold right with initial
  - `( <init> <op> ... <op> <pack-name> )` – fold left with initial
- **<op> can be most binary operators**





```cpp
template<typename...T>
int sum(T...pack) {
  return (pack + ...);
}

int main() {
  std::cout << sum(1, 2, 3, 4, 5);
}
```

Nested Namespace Definition

- **Before C++17 a namespace name in a definition could not be qualified**
  - Defining a nested namespace required opening all surrounding namespaces before

```
namespace Outer {
  namespace Middle {
    namespace Inner {
      //Declarations of Outer::Middle::inner
    }
  }
}
```

- **Since C++17 nested namespaces can be opened directly**

```
namespace Outer::Middle::Inner {
  //Declarations of Outer::Middle::Inner
}
```

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4026.html

Template Argument Deduction for Class Templates

- **In a variable declaration (or function-style cast) if the template arguments are omitted the template arguments are tried to be deduced**

```
template<typename T>
struct Box {
    Box(T content)
      : content{content}{}
    T content;
};

int main() {
  Box<int> b0{0}; //Before C++17
  Box      b1{1}; //Since C++17
}
```

- **The behavior is similar to pretending as if there was a factory function for each constructor**

```
template<typename T>
Box<T> make_box(T content) {
  return Box<T>{content};
}
```

Template Argument Deduction for Class Templates

- **What if not all factory function template parameters can be deduced by the imaginary call?**

- **Failing example (for illustration only)**
  - Here the constructor template parameter CT cannot be mapped to T by the compiler, as the type T cannot be deduced
  - Constructor template parameters are appended to the factory function template parameter list

```cpp
template<typename T>
struct Box {
    template<typename CT>
    Box(CT content)
      : content{content}{}
    T content;
};
```

```cpp
//T cannot be deduced for a
//make_box call
template<typename T, typename CT>
Box<T> make_box(CT content) {
  return Box<T>{content};
}
```

Template Argument Deduction for Class Templates

- **User-defined deduction guides can be specified in the same scope as the template**
  - Might be necessary for complex cases, e.g. template constructors if the constructor template parameters don't map directly to the class template parameters

```
<Template-Name>(<Parameter-List>) -> <Template-ID>;
```

- **Looks like a free-standing constructor**
  - Example for Box

```
template<typename T>
struct Box {
    template<typename CT>
    Box(CT content)
      : content{content}{}
    T content;
};

template<typename CT>
Box(CT) -> Box<CT>;
```

Template Argument Deduction for Class Templates

- **Example for iterator constructor**
  - The `value_type` of the iterator is extracted and used as template argument for `BoundedQueue`

```cpp
template<typename T>
struct BoundedQueue {
  template<typename Iter>
  BoundedQueue(Iter begin, Iter end);
  //...
};

template<typename Iter>
BoundedQueue(Iter, Iter) -> BoundedQueue<typename std::iterator_traits<Iter>::value_type>;

int main() {
  std::vector ints{1, 2, 3};
  BoundedQueue queue{std::begin(ints), std::end(ints)};
}
```

Structured Bindings

```
auto [<identifer-list>] = ...;
```

- **Elements of an std::tuple or public data members of a struct can be bound to multiple variables in a single declaration**

- **Number of elements to be bound must match the number of variables to introduce**

- **Reference qualifiers can be added to `auto` (& or &&)**

```
#include <iostream>
#include <tuple>

int main() {
  auto [f, s] = std::make_tuple(1, 1.5);
  std::cout << "f = " << f << '\n';
  std::cout << "s = " << s;
}
```

```
struct S {
  int member_i;
  double member_d;
};

S create();

int main() {
  auto const & [i, d] = create();
}
```

- **Article on topic:** https://skebanga.github.io/structured-bindings/

String View

- **Construction of an `std::string` object can be (relatively) expensive as it might require heap allocation for the content**

```
bool contains(std::string const & str, std::string const & substr) {
  return str.find(substr) != std::string::npos;
}

int main() {
  std::string s{"it is where you look last"};
  std::cout << std::boolalpha << contains(s, "last");
}
```

String View

- **Non-owning read operations on strings might require many overloads to be efficient when used**
  - `string_view` unifies them with a lightweight read-only wrapper

```cpp
bool contains(std::string_view str, std::string_view substr) {
  return str.find(substr) != std::string::npos;
}
```

- **Beware: std::string_views are like references and can be dangling, if the original runs out of scope!**
  - The view does not keep the data alive (non-owning)
  - Usually, std::string_views are only used for parameter types

- **Article on topic:** https://skebanga.github.io/string-view/

Further Topics to Be Covered During the Semester

- u8 character literals
- Lambda capture of `*this`
- Guaranteed copy elision
- Constexpr lambda expression
- `if constexpr` statements
- Class template argument deduction
- `std::shared_mutex`