Department I - C Plus Plus

Modern and Lucid C++ for Professional Programmers

Part 6 – ADL & Enums

IFS INSTITUTE FOR SOFTWARE

Prof. Peter Sommerlad / Thomas Corbat

Rapperswil, 24.10.2016
HS2016

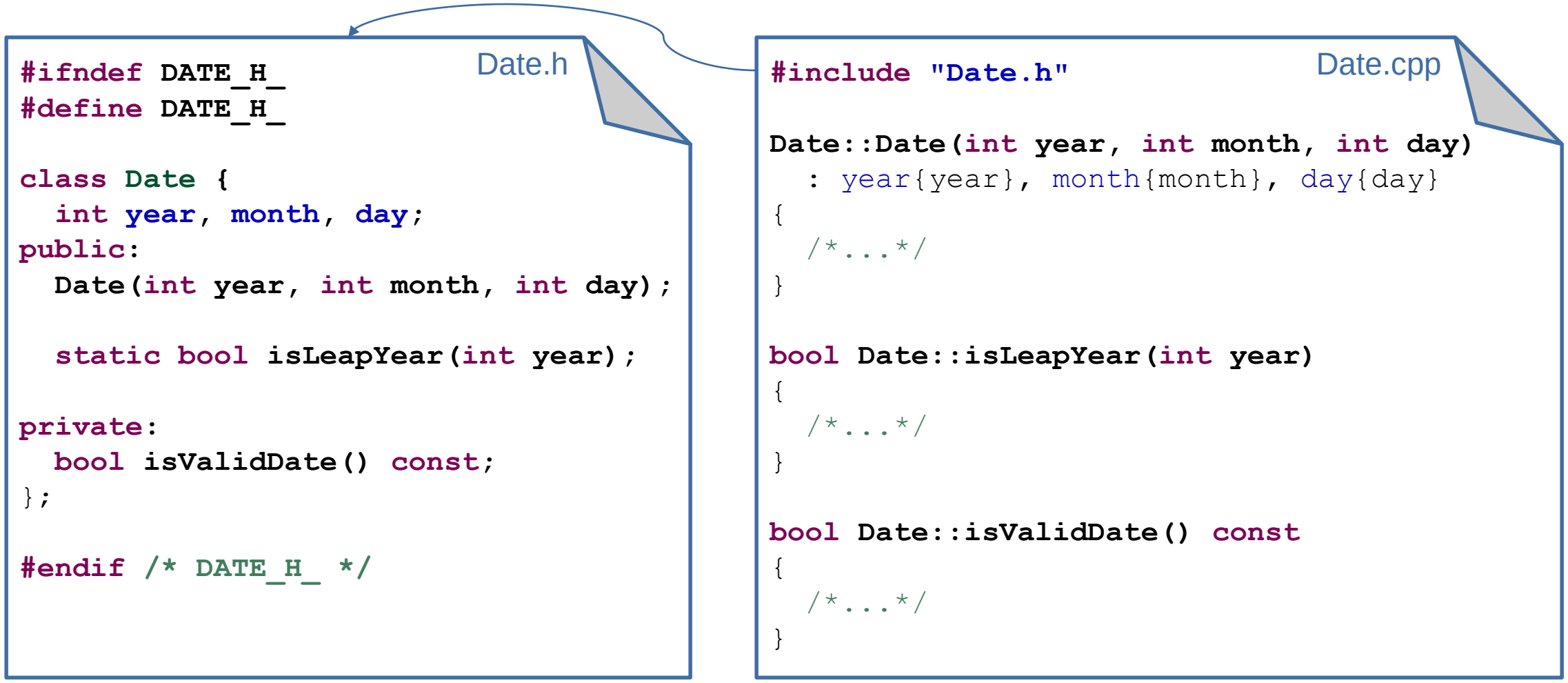
HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL
FHO Fachhochschule Ostschweiz

Warmup Week 5 – Classes & Output Operator

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Separation of Declaration and Implementation



- **Workaround `print` member function**
  - Has access to private data members
  - Can be called from free `operator<<`

Any.cpp

```cpp
#include "Date.h"
#include <iostream>

void printBirthday() {
  std::cout << Date::myBirthday;
}
```

Date.h

```cpp
#include <ostream>

class Date {
  int year, month, day;
public:
  std::ostream & print(std::ostream & os) const {
    os << year << "/" << month << "/" << day;
    return os;
  }
};
inline std::ostream & operator<<(std::ostream & os, Date const & date){
  return date.print(os);
}
```

Week 6 – Reading User Defined Types from Streams
Default Values
Argument Dependent Lookup

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

- **Input operator has the same problems as the output operator**
  - `operator>>`
  - Parameters: `std::istream &` and `Date &`
  - Returns `std::istream &` for chaining input

Any.cpp

```
#include "Date.h"
#include <iostream>

void foo() {
  Date d{};
  std::cin >> d;
}
```

Date.h

```
#include <istream>

class Date {
  int year, month, day;
public:
  std::istream & read(std::istream & is) {
    //Logic for reading values and verifying correctness
    return is;
  }
};
inline std::istream & operator>>(std::istream & is, Date & date) {
  return date.read(is);
}
```

Implementing read()

- **Expect std::istream to be in good() state as precondition and to provide a correct date**
- **If extracting a date fails set the std::istream to fail state**
- **Do not overwrite the this object if the input cannot be used to read a valid date object**
  - Keep the invariant "Date represents a valid date"

Date.h

```
//Includes

class Date {
  int year, month, day;
public:
 std::istream & read(std::istream & is) {
    int year{-1}, month{-1}, day{-1};
    char sep1, sep2;
    //read values
    is >> year >> sep1 >> month >> sep2 >> day;
    try {
      Date input{year, month, day};
      //overwrite content of this object (copy-ctor)
      (*this) = input;
      //clear stream if read was ok
      is.clear();
    } catch (std::out_of_range & e) {
      //set failbit
      is.setstate(std::ios::failbit | is.rdstate());
    }
    return is;
  }
};
```

Constructor with std::istream &

- **Declaration of a constructor that takes an std::istream & as parameter**

```
explicit Date(std::istream & in);
```

- **Declare constructors with one parameter explicit to avoid automatic conversion**
- **Throw an exception if the input does not represent a valid date**
  - For not violating the invariant
  - Alternative: Create a date with a default value

Date.h

```
#ifndef DATE_H_
#define DATE_H_

class Date {
  //...
  explicit Date(std::istream & in);
};

#endif /* DATE_H_ */
```

Date.cpp

```
#include "Date.h"

Date::Date(std::istream & in)
    : year{}, month{}, day{}
{
  read(in);
  if (in.fail())
    throw std::out_of_range{"invalid date"};
}
```

Factory Function for Creating Date Objects

- **Declaration of a factory function for Date**

```
Date make_date(std::istream & in);
```

- **Factory functions**
  - `make_xxx()` or `create_xxx()`
- **Placed in**
  - Class as static member function
  - Or in same namespace as the class
- **Delivers a default value if reading fails**
  - E.g. `Date{9999, 12, 31}`
  - Similar to std::string::npos for "not found"

```
Date make_date(std::istream & in)
try {
  return Date{in};
} catch (std::out_of_range const &) {
  return Date{9999, 12, 31};
}
```

Default Value – Default Constructor

- **As we have specified a default value, this should be the value created by the default constructor**
  - Constructor without parameters
- **Remark regarding initialization:**

```
Date nice_d{};
Date ugly_d;
```

  - Both create a Date and call the default constructor in this case
  - The second (without {}) does not work with all types. It might contain uninitialized variables
  - Good practice: Initialize all variables with {}!

Date.h

```
#ifndef DATE_H_
#define DATE_H_

class Date {
  //...
  Date();
};

#endif /* DATE_H_ */
```

Date.cpp

```
#include "Date.h"

Date::Date()
  : year{9999}, month{12}, day{31}
{
}
```

Default Initialization for Members

- **Member variables can have a default value assigned**
  - NSDMI – Non-Static Data Member Initializers

```
class <typename> {
  <type> <member>{<default-value>};
};
```

- **Such values are used if the member is not present in the initializer list of the constructor**
  - Initializer list still overrides those values
- **Useful if multiple constructors initialize date similarly**
  - Avoids duplication

Date.h

```
#ifndef DATE_H_
#define DATE_H_

class Date {
  int year{9999}, month{12}, day{31};
  //...
  Date();
  Date(int year, int month, int day);
};

#endif /* DATE_H_ */
```

Date.cpp

```
#include "Date.h"

Date::Date() {
}

Date::Date(int year, int month, int day)
  : year{year}, month{month}, day{day} {
  /*...*/
}
```

Defaulted Constructors

- **Some special member functions are implicitly available in certain cases**
  - E.g. Default constructor is implicitly available if no other explicit constructor is declared
- **(Re-)implementing the default behavior of the default constructor can be avoided by defaulting it**

```
<ctor-name>() = default;
```

  - Adds the corresponding constructor to the type with the same behavior as if it was implicitly available
  - Possible for:
    Default constructor and destructor
    Copy/move constructor
    Copy/move assignment operator

Date.h

```
#ifndef DATE_H_
#define DATE_H_

class Date {
  int year{9999}, month{12}, day{31};
  //...
  Date() = default;
  Date(int year, int month, int day);
};

#endif /* DATE_H_ */
```

Date.cpp

```
#include "Date.h"

~~Date::Date() {~~
~~}~~

Date::Date(int year, int month, int day)
  : year{year}, month{month}, day{day} {
  /*...*/
}
```

Deleted Constructors

- **Some special member functions are implicitly available in certain cases**
  - E.g. Default constructor is implicitly available if no other explicit constructor is declared
- **Those implicit constructor are not always wanted**
  - Make them explicitly private
  - Or delete them

```
<ctor-name>() = delete;
```

  - Possible for:
    Default constructor and destructor
    Copy/move constructor
    Copy/move assignment operator

Banknote.h

```
#ifndef BANKNOTE_H_
#define BANKNOTE_H_

class Banknote {
  int value;
  //...
  Banknote(Banknote & const) = delete;
};

#endif /* BANKNOTE_H_ */
```

Forger.cpp

```
#include "Banknote.h"

Banknote forge(Banknote const & note) {
  Banknote copy{note}; //!
  return copy;
}
```

Defining Date in a Namespace

- **Class `Date` can/should be put in a namespace to group it with its operators and functions**

Date.h

```
#include <iostream>

namepace date {
class Date {
  int year, month, day;
public:
  std::ostream & print(std::ostream & os) const;
};

inline std::ostream & operator<<(std::ostream & os, Date const & date) {
  return date.print(os);
}
}
```

- **Using names from a namespace requires qualification**

```
date::Date d{};
```

Defining Members for Date in a Namespace

- **Member implementations need to be qualified with that namespace too**
  - Example: `date::Date::read`

Date.h

```
namepace date {
class Date {
  int year, month, day;
public:
  std::istream & read(std::istream & is);
};
}
```

Date.cpp

```
#include "Date.h"

std::istream & date::Date::read(std::istream & is) {
  //...
  return is;
}
```

- **Types and (non-member) functions belonging to that type should be placed in a common namespace**

```
namespace date { ... }
```

- **Advantage: Argument Dependent Lookup!**
  - When the compiler encounters an unqualified function or operator call with an argument of a user-defined type it looks into the namespace in which that type is defined to resolve the function/operator
  - E.g. it is not necessary to write **std::** in front of **for_each** when std::vector::begin() is an argument of the function

date.h

```
namespace date {

class Date {
  //...
};

bool is_holiday(Date const &);

}
```

calendar.cpp

```
#include "date.h"

using Dates = std::vector<date::Date>;

void mark_holidays(Dates const & dates) {
  for_each(begin(dates), end(dates),
    [](date::Date const & d) {
      if (is_holiday(d)) //date::is_holiday
        //...
    });
}
```

Operators in Namespace – Argument Dependent Lookup

- **Unqualified function (operator) calls don't require explicit namespace in call**
  - NOT: `std::cout date::<< birthday;`
- **Functions and operators are looked up in the namespace of their arguments first**
  - E.g. namespaces `std` and `date` for `std::cout << birthday;`

Any.cpp

```
#include "Date.h"

void foo() {
  date::Date birthday{2011, 8, 2};
  //date::operator<< (std::cout, d);
  std::cout << birthday;
}
```

Date.h

```
namepace date {
class Date {
  //...
};
inline std::ostream & operator<<(std::ostream & os, Date const & date) {
  return date.print(os);
}
}
```

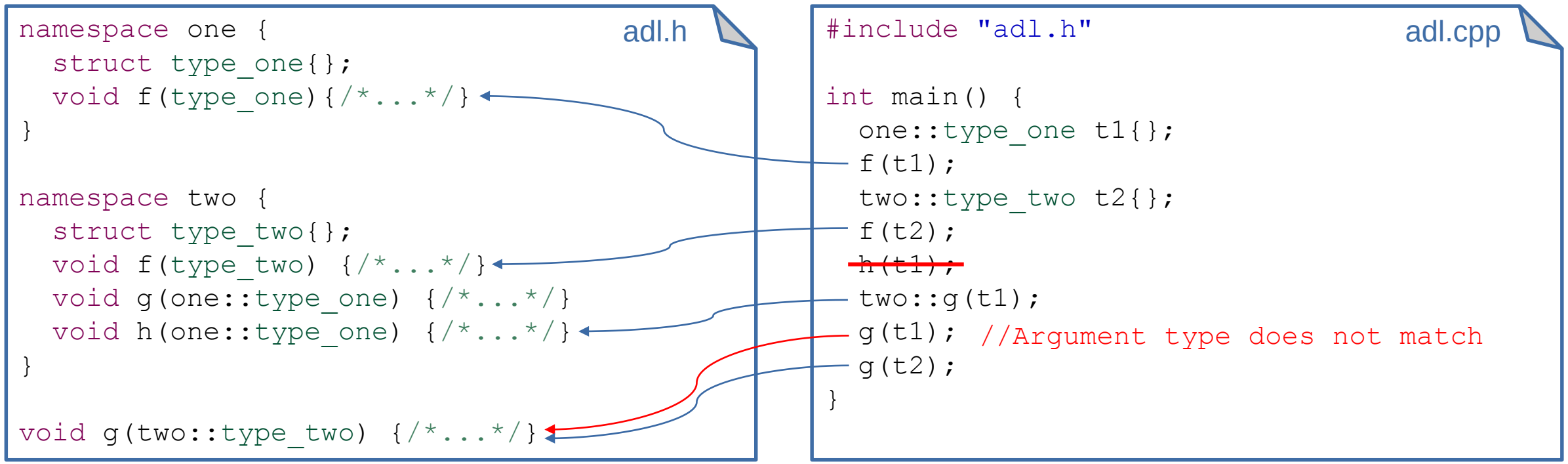
ADL Examples

adl.h

```
namespace one {
  struct type_one{};
  void f(type_one){/*...*/}
}

namespace two {
  struct type_two{};
  void f(type_two) {/*...*/}
  void g(one::type_one) {/*...*/}
  void h(one::type_one) {/*...*/}
}

void g(two::type_two) {/*...*/}
```

adl.cpp

```
#include "adl.h"

int main() {
  one::type_one t1{};
  f(t1);
  two::type_two t2{};
  f(t2);
  h(t1);
  two::g(t1);
  g(t1); //Argument type does not match
  g(t2);
}
```



ADL Issues (Where Intuition Fails)

- **Generic code (Templates) might not pick up a global `operator<<` in an algorithm call using `ostream_iterator` if the value output is from namespace `std` too**
  - E.g. `std::vector`
- **This example only works when the `operator<<(ostream &, vec const &)` is put in namespace `std`, because both arguments are in `std`**
  - This is not allowed by the standard!

intuition.cpp

```
using std::vector;
using std::ostream;
using vec = vector<int>;
using outv = std::ostream_iterator<vec>;
using out = std::ostream_iterator<int>;

namespace std {
ostream & operator<<(ostream & os, vec const & v) {
  copy(begin(v), end(v), out { os, "," });
  return os;
}
}

void work_only_with_shift_in_ns_std(ostream & os) {
  vector<vec> vv { { 1, 2, 3 }, { 4, 5, 6 } };
  copy(begin(vv), end(vv), outv { os, "\n" });
}
```

Workaround for std::-Resolution Problem

- **Create a new type by inheriting from `std::vector<int>`**
  - Requires support for inheriting constructors

```
struct Sub : Base {
  using Base::Base;
};
```

- **Alias would not be sufficient**

```
using vec = vector<int>;
```

- **Not recommended to derive from standard containers in general**
- **Does not help with std::map/std::pair**

workaround.cpp

```
using std::ostream;
using std::vector;
using out = std::ostream_iterator<int>;

namespace X {
struct vec : vector<int> { // vec is a new type
  using vector<int>::vector; // inherit ctors
};
ostream & operator<<(ostream & os, vec const & v) {
  copy(begin(v), end(v), out{os, ","});
  return os;
}
}

void works_with_inheriting_ctors(ostream & os) {
  using outv = std::ostream_iterator<X::vec>;
  vector<X::vec> vv{{1,2,3},{4,5,6}};
  copy(begin(vv), end(vv), outv{os, "\n"});
}
```

Week 6 – Enums

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

- **Enumerations are useful to represent types with only a few values**

```
enum [class] <name> {
  <enumerators>
};
```

- **An enumeration creates a new type that can easily be converted to an integral type**
  - Conversion from an integral type to an enumeration is not possible directly
- **The individual values (enumerators) are specified in the type**
- **Unless specified explicitly the values start with 0 and increase by 1**

- **Unscoped enumeration (no class keyword)**

```
enum day_of_week {
  Mon, Tue, Wed, Thu, Fri, Sat, Sun
}; 0   1   2   3   4   5   6
```

  - Enumerators leak into surrounding scope
  - Best used as member of a class
- **Scoped enumeration (class keyword)**

```
enum class day_of_week {
  Mon, Tue, Wed, Thu, Fri, Sat, Sun
}; 0   1   2   3   4   5   6
```

  - Enumerators do not leak into surrounding scope
  - Underlying type can be specified

- **Unscoped enumeration (no class keyword)**
  - Enumerators leak into surrounding scope

```
namespace date {

enum day_of_week {
  Mon, Tue, Wed, Thu, Fri, Sat, Sun
};

}

bool is_weekend(date::day_of_week day)
{
  return day == date::Sat ||
         day == date::Sun;
}
```

- **Scoped enumeration (class keyword)**
  - Enumerators do not leak into surrounding scope

```
namespace date {

enum class day_of_week {
  Mon, Tue, Wed, Thu, Fri, Sat, Sun
};

}

bool is_weekend(date::day_of_week day)
{
  return day == date::day_of_week::Sat ||
         day == date::day_of_week::Sun;
}
```

Enumeration Conversion

- **Operators can be overloaded for `enum`s**
- **Examples**
  - Prefix increment

```
dayOfWeek operator++(dayOfWeek &)
```

  - Postfix increment

```
dayOfWeek operator++(dayOfWeek &, int)
```

- **Implicit conversion from `enum` to `int`**
- **Explicit conversion from `int` to `enum`**

```
dayOfWeek tuesday =
static_cast<dayOfWeek>(1);
```

```
int day = Sun;
```

```
enum dayOfWeek {
  Mon, Tue, Wed, Thu, Fri, Sat, Sun
};

dayOfWeek operator++(dayOfWeek & aday) {
  int day = (aday + 1) % (Sun + 1);
  aday = static_cast<dayOfWeek>(day);
  return aday;
}

dayOfWeek operator++(dayOfWeek & aday, int) {
  dayOfWeek ret{aday};
  if (aday == Sun)
    aday = Mon;
  else
    aday = static_cast<dayOfWeek>(aday + 1);
  return ret;
}
```

Defining Values of Enumerators

- **With = values can be specified for enumerators**
  - Subsequent enumerators get value incremented (+1)
- **Different enumerators can have the same value**
- **In some cases enumerations are used to create bit masks**
  - Values are a power of 2 (x^2)

```
enum month {
  jan = 1, feb, mar, apr, may,
  jun, jul, aug, sep, oct, nov, dec,
  january = jan, february, march,
  april, june = jun, july, august,
  september, october, november,
  december
};
```

```
enum file_permissions {
  readable = 1,
  writeable = 2,
  executable = 4
};
```

- **Enumerator names are not mapped automatically to their original name**
  - Might become a feature in a future C++ standard (2020?)
- **You can provide a lookup table and overload the output operator (<<)**

```
std::ostream & operator<<(std::ostream & out, month m) {
  static std::array<std::string, 12> const month_names {
    "Jan", "Feb", "Mar", "Apr", "May", "Jun",
    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
  out << month_names[m - 1];
  return out;
}
```

Specifying the Underlying Type

- **Enumeration classes can specify the underlying type by inheritance**
- **The underlying type can be any integral type**
- **This allows forward-declaring enumerations**
  - Can be used to hide implementation details if defined as a class member

```
enum class launch_policy
 : unsigned char
{
  sync = 1,
  async = 2,
  gpu = 4,
  process = 8,
  none = 0
};
```

Enumeration Example


```
#ifndef STATEMACHINE_H_
#define STATEMACHINE_H_

struct Statemachine {
  Statemachine();
  void processInput(char c);
  bool isDone() const;
private:
  enum class State : unsigned short;
  State theState;
};

#endif /* STATEMACHINE_H_ */
```

```
#include "Statemachine.h"
#include <cctype>
enum class Statemachine::State : unsigned short {
  begin, middle, end
};
Statemachine::Statemachine()
  : theState {State::begin} {}

void Statemachine::processInput(char c) {
  switch (theState) {
    case State::begin:
      if (!isspace(c)) theState = State::middle;
      break;
    case State::middle:
      if (isspace(c)) theState = State::end;
      break;
    case State::end:
      break; // ignore input
  }
}

bool Statemachine::isDone() const {
  return theState == State::end;
}
```

Week 6 – Arithmetic Types (Ring5)

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

- **Disclaimer: You usually do not want to implement your own arithmetic types!**
  - Future C++ standards will provide additional arithmetic types, like unbounded large integer and rational
- **Here are the basics if you ever need them anyway**
- **Example: Modulo arithmetic**
  - Rings / Finite fields
  - Used in Cryptography

- **Ring5: Arithmetic Modulo 5**

```
struct Ring5 {
  explicit Ring5(unsigned x = 0u)
    : val{x % 5}{}
  unsigned value() const {
    return val;
  }
private:
  unsigned val;
};
```

- **Invariant:**
  - Member variable is in range 0..4
- **Accessor to value**
- **Constructor explicit?**

- **Arithmetic types must be equality comparable**
  - operator ==
  - CUTE requires this operator in ASSERT_EQUAL

```
void testValueCtorWithLargeInput() {
  Ring5 four{19};
  ASSERT_EQUAL(Ring5{4}, four);
}
```

- **Boost can be used to get != easily**
  - Use boost::equality_comparable

```
struct Ring5 :
    boost::equality_comparable<Ring5> {
  bool operator==(Ring5 const & r) const {
    return val == r.val;
  }
  //...
};
```

Output Operator for Ring5

- **It might be convenient to have the output operator (<<) to print a Ring5**
  - As always: Not as a class member!
- **CUTE requires the output operator for nice failure messages**

```
std::ostream & operator<<(
    std::ostream & out,
    Ring5 const & r) {
  out << "Ring5{" << r.value() << '}';
  return out;
}
```

```
void testOutputOperator() {
  std::ostringstream out{};
  out << Ring5{4};
  ASSERT_EQUAL("Ring5{4}", out.str());
}
```

Plus Operation for Ring5

- **Result must be in the range 0..4**
  - E.g. (4 + 4) = 8 % 5 = 3
- **Implement both operators yourself**
  - operator+
  - operator+=
- **(Or) Use boost**
  - Implement operator+=
  - Derive from boost::addable<Ring5>

```
struct Ring5 :
    boost::equality_comparable<Ring5>,
    boost::addable<Ring5> {
  Ring5 operator+=(Ring5 const & r) {
    val = (val + r. val) % 5;
    return *this;
  }
  //...
};
```

```
void testAdditionWrap() {
  Ring5 four{4};
  Ring5 three = four + four;
  ASSERT_EQUAL(Ring5{3}, three);
}
```

Multiplication Operation for Ring5

- **Example for self implemented multiplication**
  - Operator *= as member function
  - Operator * as free (inline) function
    - Pass left-hand operand by value
- **Modulo only needed in *= Operator**
  - Avoids code duplication

```
struct Ring5 :
    boost::equality_comparable<Ring5>,
    boost::addable<Ring5> {
  Ring5 operator*=(Ring5 const & r) {
    val = (val * r. val) % 5;
    return *this;
  }
  //...
};
```

```
inline Ring5 operator*(
    Ring5 l, Ring5 const & r) {
  l *= r;
  return l;
}
```

Mixed Arithmetic with Ring5

- **What if we want to add Ring5 and int?**
- **Either implement all combinations of parameters for operator+**
  - operator+(Ring5, int)
  - operator+(int, Ring5)
- **Or make constructor non-explicit**
- **The latter might become a problem when we want to have automatic conversion to unsigned as well**
  - Ambiguity or wrong conversion happens

- **`four` should have the value 4**
- **`four` should have the type Ring5**

```
void test_AdditionWithInt_ValueIsFour() {
  Ring5 two{2};
  auto four = two + 2u;
  ASSERT_EQUAL(Ring5{4}, four);
}

void test_AdditionWithInt_TypeIsRing5() {
  Ring5 two{2};
  auto four = two + 2u;
    ASSERT_EQUAL(typeid(Ring5).name(),
      typeid(decltype(four)).name());
}
```

- **Overhead with code, duplication, danger of wrong implementation**
  - Test cases help
  - Overload could be forgotten

```
inline Ring5 operator+(Ring5 const & l, unsigned r) {
  return Ring5{l.value() + r};
}

inline Ring5 operator+(unsigned l, Ring5 const & r) {
  return Ring5{l + r.value()};
}
```

- **Non-explicit constructor provides automatic inward conversion**
  - Type conversion operator
    - `operator <type>() const` member function
    - explicit preferred but requires static_cast
- **Danger: Ambiguities and unexpected conversions are lurking**

```
struct Ring5 {
  Ring5(unsigned x) : val{ x % 5 } {}
  operator unsigned() const {
    return val;
  }
  //...
};
```

- **C++ allows to mark functions and constructors as constexpr**
- **Guarantees compiler to evaluate function**
- **Provides support for literals type that can be initialized at compile-time**
- **Functions must be "pure" functions**
  - no external side-effects
  - no memory
  - no exceptions
- **Faster program execution**

```
struct Ring5 {
  explicit constexpr Ring5(unsigned x)
    : val{ x % 5 } {
  }
  constexpr Ring5 operator+(Ring5 const & r) const {
    return Ring5{val + r.val};
  }
  //...
};
```

- **Writing `Ring5{4}` to create a value of type Ring5 is a bit annoying when many such values are needed**
- **`4_R5` would be shorter**
- **User-defined literals allow to defined suffixes to constants to change their type or value**

```
constexpr <type> operator"" <suffix>(<parameter>);
```

- **Must be put in a (separate) namespace**

```
namespace R5 {
  constexpr Ring5 operator"" _R5(unsigned long long v) {
    return Ring5{static_cast<unsigned>(v % 5)};
  }
}
```

  - Requires using namespace directive

```
using namespace R5;
static_assert(Ring5{2} == 7_R5, "UDL operator");
```