

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, 25.10.2017
HS2017



HSR

HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Warmup – Classes & Output Operator

- Class definition in header file
- Member implementation in source file

```
Date.h  
  
#ifndef DATE_H_  
#define DATE_H_  
  
class Date {  
    int year, month, day;  
public:  
    Date(int year, int month, int day);  
  
    static bool isLeapYear(int year);  
  
private:  
    bool isValidDate() const;  
};  
  
#endif /* DATE_H_ */
```

```
Date.cpp  
  
#include "Date.h"  
  
Date::Date(int year, int month, int day)  
    : year{year}, month{month}, day{day} {  
    /*...*/  
}  
  
bool Date::isLeapYear(int year) {  
    /*...*/  
}  
  
bool Date::isValidDate() const {  
    /*...*/  
}
```

- **The default constructor is available implicitly**
 - Unless another constructor is implemented
- **You usually don't need to implement**
 - Copy/Move Constructors
 - Destructor
- **Use the member initializer list!**
- **Constructors with one parameter should be declared explicit**

```
                                     Date.h
#ifndef DATE_H_
#define DATE_H_

class Date {
    int year, month, day;
public:
    Date(int year, int month, int day)
        : year{year}, month{month}, day{day} { /*...*/ }

    static bool isLeapYear(int year) { /*...*/ }

private:
    bool isValidDate() const { /*...*/ }
};

#endif /* DATE_H_ */
```

- **Input (>>) and Output (<<) always as free operator!**

- With read/print helper member function

```
class Date { Date.h
    int year, month, day;
public:
    std::istream & read(std::istream & is);
    std::ostream & print(std::ostream & os) const ;
};
std::istream & operator>>(std::istream & is, Date & date);
std::ostream & operator<<(std::ostream & os, Date const & date);
```

```
#include "Date.h" Date.cpp

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

Namespaces

Goals:

- You know to group and structure names into namespaces
- You know how Argument Dependent Lookup works

- **Namespaces are scopes for grouping and preventing name clashes**
 - Same name in different scopes possible
 - Example `boost::optional` and `std::optional` can coexist
- **Global namespace has the `::` prefix**
 - Can be omitted if unique
 - `::std::cout` is usually equal to `std::cout`
- **Nesting of namespaces is possible**
 - Example: `std::literals::chrono_literals`
- **Nesting of scopes allows hiding of names**
 - Hidden names can only be accessed when they belong to a named scope

- Namespaces can only be defined outside of classes and functions
- The same namespace can be opened and closed multiple times to gather definitions and declarations (header files)
- Qualified names are used to access names in a namespace
 - `demo::subdemo::foo()`
- A name with a leading `::` is called a fully qualified name
 - `::std::cout`

```
namespace demo {
void foo(); //1
namespace subdemo {
void foo() { /*2*/ }
} // subdemo
} // demo

namespace demo {
void bar() {
    foo(); //1
    subdemo::foo(); //2
}
}

void demo::foo() { /*1*/ } // definition

int main() {
    using demo::subdemo::foo;
    foo(); //2
    demo::foo(); //1
    demo::bar();
}
```


- **Import a name from a namespace into the current scope**

- That name can be used without a namespace prefix
- Useful if the name is used very often

```
using std::string;  
string s{"no std::"};
```

- **Alternative: Using alias for types if a name is long**

```
using input = std::istream_iterator<int>;  
input eof{};
```

- **There are also using directives, which import ALL names of a namespace into the current scope**

- Use them only in local scope to avoid "pollution" of your namespace

```
int main() {  
    using namespace std;  
    cout << "Hello John";  
    //many more uses of std names  
}
```

- **Special case: omit name after namespace**
 - The compiler chooses a unique internal identifier
- **Implicit using directive for the chosen name**
- **Hides module internals**
 - Helper functions and types
 - Constants
- **Use them only in source files (*.cpp)**

```
#include <iostream>
namespace { // anonymous
void doit() { // can not be called
              // outside this file
    std::cout << "doit called\n";
}
} // anonymous namespace ends

void print() { // callable from other
              // parts if declared
    doit();
    std::cout << "print called\n";
}
```

```
void caller() { // in a different file
    void print(); // declare print
    print();
    void doit(); // declare doit
    doit();      // linker error ⚡
}
```

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

```
namespace 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);  
}  
}
```

Date.h

- Using names from a namespace requires qualification

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

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

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

Date.h

```
#include "Date.h"

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

Date.cpp

- 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

```
namespace date { date.h  
  
class Date {  
    //...  
};  
  
bool is_holiday(Date const &);  
  
}
```

```
#include "date.h" calendar.cpp  
  
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  
                    //...  
            });  
}
```

- Unqualified operator calls don't allow explicit namespace qualification
 - NOT: `std::cout date::<< birthday;` ⚡
- Functions and operators are looked up in the namespace of the type of their arguments first
 - E.g. namespaces `std` and `date` for `std::cout << birthday;`

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

```
#include "Date.h"                                     Any.cpp  
  
void foo() {  
    date::Date birthday{2011, 8, 2};  
    //date::operator<< (std::cout, d);  
    std::cout << birthday;  
}
```

Date.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.h

```
#include "adl.h"

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

adl.cpp

- **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<int>`
- **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"});
}
```


- 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`

```
using std::ostream;                                workaround.cpp  
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"});  
}
```

Enums

Goals:

- You know how to create enumerations as simple types with few values
- You know the difference between scoped and unscoped enumerations
- You know how to specify the values of enumerators

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

```
enum <name> {  
    <enumerators>  
};
```

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

- An enumeration creates a new type that can easily be converted to an integral type (for unscoped enumeration)
 - Conversion from an integral type to an enumeration is not possible implicitly
- 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  
};  
    //Enumerators are visible here  
}  
  
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;  
}
```

- Operators can be overloaded for **enums**

- Examples

- Prefix increment

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

- Postfix increment

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

- Implicit conversion from **enum** to **int**
(Only for unscoped enums)

```
int day = Sun;
```

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

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

```
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;
}
```

- **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]; //m - 1 if Jan has value 1  
    return out;  
}
```

- 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
};
```



```
#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;
}
```

Arithmetic Types (Ring5)

Goal:

- You know how to implement your own arithmetic type
- You know the possible ambiguities that might arise from implicit conversion

- **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;  
    }  
    //...  
};
```

- It might be convenient to have the output operator (<<) to print a Ring5
 - As always: Output operator 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());  
}
```

- **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);  
}
```

■ 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;  
}
```

- What if we want to add Ring5 and int?
- Either implement all combinations of parameters for operator+
 - operator+(Ring5, unsigned)
 - operator+(unsigned, 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");
```