Concepts of C++ Programming

Lecture 5: Classes and Conversions

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2024/25

static_assert[59]

- `static_assert(bool expr, string)` – assert at compile-time
- Expression must be a compile-time constant
- Can have an optional failure message

Example:

```
static_assert(sizeof(int) == 4, "program only works on 4-byte integers");
```

[59] https://en.cppreference.com/w/cpp/language/static_assert

Classes

```cpp
class Name1 {
    // member specifications...
};
struct Name2 {
    // member specifications...
};
```

- Name can be any valid identifier
- Members can be:
  - Variables (data members)
  - Functions (member functions)
  - Types (nested types)
- Note the trailing semicolon

Data Members[60]

- Declarations of (non-`extern`) variables
- Size of declared variable must be known (see later)
- Variable name must be unique within class
- Variables can have default value

```
class Name {
    int foo = 10;
    int& iref;
    float* ptr;
    const char x;
};
```

[60] https://en.cppreference.com/w/cpp/language/data_members

Data Layout

- Class is essentially just a sequence of its data members

- Members are stored in memory in declaration order
- Alignment of members is respected $\rightsquigarrow$ padding between objects
- Alignment of class is largest alignment of data members

```
class C {
    int i; // sizeof = 4; alignof = 4; offset = 0
    // (4 padding bytes)
    int* p; // sizeof = 8; alignof = 8; offset = 8
    char c; // sizeof = 1; alignof = 1; offset = 16
    // (2 padding bytes)
    short s; // sizeof = 2; alignof = 2; offset = 18
    // (4 padding bytes -- sizeof must be multiple of alignof)
}; // sizeof(C) = 24; alignof(C) == 8
```

Data Layout

Quiz: What is the size of Line?

```
class Point {
    int x;
    int y;
    unsigned char color;
};
class Line {
    Point a;
    Point b;
    unsigned char lineWidth;
};
```

A. (compile error) B. 19 C. 24 D. 28 E. 32

Bit Fields[61]

- Can specify bit-size for integer members
- Adjacent bit fields packed together
- Access is fairly expensive, but might reduce memory usage

⇝ Use only when strongly beneficial

```
class Bitfields {
    unsigned short flagA : 1;
    unsigned short flagB : 1;
    unsigned short tinyVar : 11;
};
static_assert(sizeof(Bitfields) == 2);
static_assert(alignof(Bitfields) == 2);
```

[61] https://en.cppreference.com/w/cpp/language/bit_field

Data Layout

Quiz: What is the size of this class?

```
class Value { // (excerpt from llvm/include/llvm/IR/Value.h)
  const unsigned char SubclassID;
  unsigned char HasValueHandle : 1;
  unsigned char SubclassOptionalData : 7;
  unsigned short SubclassData;
  unsigned NumUserOperands : 27;
  unsigned IsUsedByMD : 1;
  unsigned HasName : 1;
  unsigned HasMetadata : 1;
  unsigned HasHungOffUses : 1;
  unsigned HasDescriptor : 1;
  Type *VTy;
  Use *UseList;
}; // NB: sizeof(void*) == 8; sizeof(unsigned) == 4
```

A. (compile error) B. 24 C. 32 D. 40 E. 45

Data Layout: Consequences

- Order of members has impact on class size

$\Rightarrow$ When class size is important, reduce padding

$\Rightarrow$ Recommendation: place all data members together at beginning/end

  - Potential padding etc. is easily findable

- All users of the class need to know the declaration

$\Rightarrow$ Class declarations often put in header files

$\Rightarrow$ Adding/modifying members requires changes data layout $\Rightarrow$ recompilation

  - Especially important when distributing libraries – all users *must* rebuild

Member Functions

- Declaration of methods just like regular function declarations
- Inline definitions are implicitly `inline`
- Out-of-line definitions are preferable for non-trivial methods

```
//--- foo.h
#pragma once
class Foo {
    int foo();
    int bar(int x) { // inline definition
        return x + 1;
    }
};
//--- foo.cpp
int Foo::foo() { // out-of-line definition
    return 10;
}
```

Inline vs. Out-Of-Line Definitions

Quiz: Which answer is NOT correct?

A. Out-of-line definitions tend to allow for more optimizations.
B. Out-of-line definitions tend to reduce compile time.
C. Inline definitions tend to allow for more optimizations.
D. Inline definitions in headers are possibly compiled several times.

- Similar considerations as for `inline` functions apply

Member Access

```
struct Vec {
    unsigned x;
    unsigned y;
};
Vec v;
Vec* vp = ...;

// member access:
int l1dist_a = v.x + v.y;
// ptr->member is a shorthand for (*ptr).member
int l1dist_b = vp->x + vp->y;
```

this

- Member functions have implicit parameter `this`; type is `Class*`
- In member functions, members can be accessed without `this` (preferred)

```
struct Vec {
    unsigned x;
    unsigned y;

    unsigned l1dist() {
        return this->x /* explicit access */ + y /* implicit access*/;
    }
};
Vec v;
Vec* vp = ...;
int l1dist_a = v.l1dist();
int l1dist_b = vp->l1dist();
```

const-Qualified Member Functions

- Member functions can be const-qualified
- this is a const Class*

⇒ Data members are immutable

```
struct Vec {
    unsigned x;
    unsigned y;
    unsigned getX() const { return x; }
    unsigned getY() const { return y; }
    unsigned l1dist() const;
};
unsigned Vec::l1dist() const {
    return x + y; // this is a const Vec*
}
```

Constness and Member Functions

- For *non-const lvalues* non-const overloads are preferred over const ones
- For *const lvalues* only const-qualified functions are selected

```cpp
struct Foo {
  int getA() { return 1; }
  int getA() const { return 2; }
  int getB() const { return getA(); }
  int getC() { return 3; }
};
Foo& foo = /* ... */;
const Foo& cfoo = /* ... */;
```

Expression	Value
`foo.getA()`	1
`foo.getB()`	2
`foo.getC()`	3
`cfoo.getA()`	2
`cfoo.getB()`	2
`cfoo.getC()`	*error*

Constness of Member Variables

- Constness propagates through pointer lvalue access
- `const` data members are always constant
  - Can only be set once during construction (see later)
- `mutable` member variables are always non-const (use carefully!)

```
struct Foo {
    int i;
    const int c;
    mutable int m;
}
Foo& foo = /* ... */;
const Foo& cfoo = /* ... */;
```

Expression	Value Category
`foo.i`	non-const lvalue
`foo.c`	const lvalue
`foo.m`	non-const lvalue
`cfoo.i`	const lvalue
`cfoo.c`	const lvalue
`cfoo.m`	non-const lvalue

Static Members[62]

- Static data members: members not bound to class instances
- Only one instance in the program, like global variables
- Static member functions: no implicit `this` parameter
- Static members can be accessed with `::` operator

```
//--- foo.h
struct Foo {
    static int var; // declaration
    static void statfn(); // declaration
};
//--- foo.cpp
int Foo::var = 10; // definition
void Foo::statfn() { /* ... */ } // definition
```

[62] https://en.cppreference.com/w/cpp/language/static

Constructors

- ... are special functions that are called when an object is *initialized*
- ... have no return type, no const-qualifier, and name is class name
- ... can have arguments, constructor without arguments is *default constructor*
- ... are sometimes implicitly defined by the compiler

```cpp
struct Foo {
    Foo() {
        // default constructor
    }
};
```

```cpp
struct Foo {
    int a;
    Bar b;
    // Default constructor is
    // implicitly defined, does
    // nothing with a, calls
    // default constructor of b
};
```

Initializer List

- Specify how member variables are initialized before constructor body
- Other constructors can be called in the initializer list
- Members initialized in the order of their definition
- `const` member variables can only be initialized in the initializer list

```
struct Foo {
    int a = 123; float b; const char c;
    // default constructor initializes a (to 123), b, and c
    Foo() : b(2.5), c(7) {}
    // initializes a and b to the given values
    Foo(int a, float b, char c) : a(a), b(b), c(c) {}
    Foo(float f) : Foo() {
        // First the default constructor is called, then the body
        // of this constructor is executed
        b *= f;
    }
};
```

Initializing Objects[63]

- Constructor executed on initialization
- Arguments given in the initialization are passed to the constructor
- C++ has several types of initialization that are very similar but unfortunately have subtle differences:
  - *default initialization* (`Foo f;`)
  - *value initialization* (`Foo f{};` and `Foo()`)
  - *direct initialization* (`Foo f(1, 2, 3);`)
  - *list initialization* (`Foo f{1, 2, 3};`)
  - *copy initialization* (`Foo f = g;`)
- Simplified syntax: *class-type identifier(arguments)*`;` or *class-type identifier{arguments}*`;`

[63] https://en.cppreference.com/w/cpp/language/initialization

Constructors (1)

Quiz: What is the output of the following program?

```
#include <print>
struct Foo {
    int answer;
    Foo() : answer(42) {}
};
int main() {
    Foo f();
    std::println("{}", f.answer);
    return 0;
}
```

A. (compile error) B. 0 C. 42 D. (undefined behavior)

Constructors (2)

Quiz: What is the return value of foo?

```
struct C {
    int i;
    C() = default;
};
int foo() {
    const C c;
    return c.i;
}
```

A. (compile error)   B. an arbitrary integer   C. 0   D. (undefined behavior)

Constructors (3)

Quiz: What is problematic about this program?

```
#include <print>
struct Foo {
    const int& answer;
    Foo() {}
    Foo(const int& answer)
        : answer(answer) {}
};
```

```
int main() {
    int answer = 42;
    Foo f(answer);
    std::println("{}", f.answer);
    return 0;
}
```

A. Compile error: Two constructors are not allowed.

B. Compile error: `answer` not always initialized.

C. Compile error: `f` is a function declaration.

D. Undefined behavior: `f.answer` is a dangling reference.

E. There is no problem: the program always prints 42.

Constructors (4)

Quiz: What is problematic about this program?

```
#include <print>
struct Foo {
    const int& answer;
    Foo(const int& answer)
        : answer(answer) {}
};
```

```
int main() {
    int answer = 42;
    Foo f = answer;
    std::println("{}", f.answer);
    return 0;
}
```

A. Compile error: Cannot assign integer to type `Foo`.

B. Compile error: Cannot convert integer to `Foo`.

C. Undefined behavior

D. There is no problem: the program always prints 42.

Converting and Explicit Constructors[64]

- Constructors with one argument used for *explicit* and *implicit conversions*
- Use `explicit` to disallow implicit conversion
- Generally, use `explicit` unless there's a good reason not to

```cpp
struct Foo {
    Foo(int i);
};
void print_foo(Foo f);
// Implicit conversion,
// calls Foo::Foo(int)
print_foo(123);
// Explicit conversion,
// calls Foo::Foo(int)
static_cast<Foo>(123);
```

```cpp
struct Bar {
    explicit Bar(int i);
};
void print_bar(Bar f);
// Implicit conversion,
// compiler error!
print_bar(123);
// Explicit conversion,
// calls Bar::Bar(int)
static_cast<Bar>(123);
```

[64] https://en.cppreference.com/w/cpp/language/converting_constructor

Member Access Control

- Every member has `public`, `protected` or `private` access
- Default for `class`: `private`; for `struct`: `public`
  - Recommendation: always explicitly specify access control
- `public` = accessible by everyone, `private` only by class itself

```
class Foo {
    int a; // a is private
public: // All following declarations are public
    int b;
    int getA() const { return a; }
protected: // All following declarations are protected
    int c;
public: // All following declarations are public
    static int getX() { return 123; }
};
```

Friend Declarations[65]

- Class body can contain *friend declarations*
- Friend: has access to private/protected members

- `friend function-declaration;` (for friend function)
- `friend class-specifier;` (for friend class)

```
class A {
    int a; // private
    friend class B;
    friend void foo(A&);
};
class B {
    void bar(A& a) {
        a.a = 42; // OK
    }
};
```

```
class C {
    void foo(A& a) {
        a.a = 42; // ERROR
    }
};
void foo(A& a) {
    a.a = 42; // OK
}
```

[65] https://en.cppreference.com/w/cpp/language/friend

Nested Types

- For nested types classes behave just like a namespace
- Nested types are accessed with `::`
- Nested types are `friends` of their parent

```
struct A {
    struct B {
        int getI(const A& a) {
            return a.i; // OK, B is friend of A
        }
    };
private:
    int i;
};
A::B b; // reference nested type B of class A
```

Forward Declarations

- Classes can be forward declared: `class Name;`
- Type is *incomplete* until defined later
- Incomplete type can be used, e.g., for pointer/reference

```
//--- foo.h
class A;
class ClassFromExpensiveHeader;
class B {
    ClassFromExpensiveHeader* member;
    void foo(A& a);
};
class A {
    void foo(B& b);
};
//--- foo.cpp
#include "ExpensiveHeader.hpp"
// ...
```

Incomplete Types[66]

- No operations that require size/layout of type are possible
  - No pointer arithmetic
  - No access to members, member functions, etc.
  - No definition/call of function with incomplete return/argument type
- Sometimes, this information is not needed:
  - E.g., pointer/reference declarations can refer to incomplete types
  - E.g., member functions with incomplete parameter types

[66] https://en.cppreference.com/w/cpp/language/types#Incomplete_type

Operator Overloading[67]

- Classes can overload built-in operators like +, ==, etc.
- Many overloaded operators can also be written as non-member functions
- Overloaded operators are selected with the regular overload resolution
- Overloaded operators are not required to have meaningful semantics

- Almost all operators can be overloaded, exceptions are: `::`, `.`, `.*`, `?:`
- This includes "unusual" operators like:
  = (assignment), `()` (call), $*$ (dereference), & (address-of), `,` (comma)

[67] https://en.cppreference.com/w/cpp/language/operators

Arithmetic Operators[68]

$lhs\ op\ rhs \sim lhs.\text{operator}\ op(rhs)$ or $\text{operator}\ op(lhs,\ rhs)$

- Overloaded versions of `||` and `&&` lose their special behaviors
- Should be `const` and take `const` references
- Usually return a value and not a reference

```
struct Int {
    int i;
    Int operator+(const Int& other) const { return Int{i + other.i}; }
    Int operator-() const { return Int{-i}; };
};
Int operator*(const Int& a, const Int& b) { return Int{a.i * b.i}; }
Int a{123}; Int b{456};
a + b; /* is equivalent to */ a.operator+(b);
a * b; /* is equivalent to */ operator*(a, b);
-a; /* is equivalent to */ a.operator-();
```

[68] https://en.cppreference.com/w/cpp/language/operator_arithmetic

Comparison Operators[69]

All binary comparison operators (`<`, `<=`, `>`, `>=`, `==`, `!=`, `<=>`) can be overloaded.

- Should be `const` and take `const` references
- Return `bool`, except for `<=>` (see next slide)
- If only `operator<=>` is implemented, `<`, `<=`, `>`, and `>=` work as well
- `operator==` must be implemented separately (then `!=` works, too)

```
struct Int {
    int i;
    std::strong_ordering operator<=>(const Int& a) const {
        return i <=> a.i;
    }
    bool operator==(const Int& a) const { return i == a.i; }
};
Int a{123}; Int b{456};
a < b; /* is equivalent to */ (a.operator<=>(b)) < 0;
a == b; /* is equivalent to */ a.operator==(b);
```

[69] https://en.cppreference.com/w/cpp/language/operator_comparison

Three-Way[70]

`operator<=>` should return one of the following types from `<compare>`: `std::partial_ordering`, `std::weak_ordering`, `std::strong_ordering`.

- When comparing two values `a` and `b` with `ord = (a <=> b)`, then `ord` has one of the three types and can be compared to 0:
- `ord == 0` $\Leftrightarrow$ `a == b`
- `ord < 0` $\Leftrightarrow$ `a < b`
- `ord > 0` $\Leftrightarrow$ `a > b`
- `strong_ordering` convertible to `weak_ordering` and `partial_ordering`
- `weak_ordering` convertible to `partial_ordering`

[70] https://en.cppreference.com/w/cpp/utility/compare/partial_ordering

Three-Way Comparison (2)

- `partial_ordering` can be unordered, i.e. neither `a <= b` nor `a >= b`
  - `std::partial_ordering::less`, `::equivalent`, `::greater`, `::unordered`
  - Example: floating-point numbers, NaN is unordered
- `std::weak_ordering` or `std::strong_ordering` for total order
  - `::less`, `::equivalent`, `::greater`
  - `strong_ordering`: equal values must be completely indistinguishable
  - Example for strong ordering: integers
  - Example for weak ordering: points in 2d-space ordered by distance from origin

Increment and Decrement[71]

Pre- and post-inc/dec are distinguished by an (unused) `int` argument

- `C& operator++(); C& operator--();`
  pre-increment or -decrement, modify object, return `*this`
- `C operator++(int); C operator--(int);`
  post-increment or -decrement, copy self, modify self, return unmodified copy

```
struct Int {
    int i;
    Int& operator++() { ++i; return *this; }
    Int operator--(int) { Int copy{*this}; --i; return copy; }
};
Int a{123};
++a; // a.i is now 124
a++; // ERROR: post-increment is not overloaded
Int b = a--; // b.i is 124, a.i is 123
--b; // ERROR: pre-decrement is not overloaded
```

[71] https://en.cppreference.com/w/cpp/language/operator_incdec

Subscript Operator[72]

Classes behaving like containers/pointers usually override the *subscript* `[]`

- `a[b]` is equivalent to `a.operator[](b)`
- Type of `b` can be anything, for array-like containers it is usually `size_t`

```
struct Foo { /* ... */ };
struct FooContainer {
    Foo* fooArray;
    Foo& operator[](size_t n) { return fooArray[n]; }
    const Foo& operator[](size_t n) const { return fooArray[n]; }
};
```

[72] https://en.cppreference.com/w/cpp/language/operator_member_access

Dereference Operators[73]

Classes behaving like pointers usually override the operators * and ->

- operator*() usually returns a reference
- operator->() should return a pointer or an object that itself has an overloaded -> operator

```
struct Foo { /* ... */ };
struct FooPtr {
    Foo* ptr;
    Foo& operator*() { return *ptr; }
    const Foo& operator*() const { return *ptr; }
    Foo* operator->() { return ptr; }
    const Foo* operator->() const { return ptr; }
};
```

[73] https://en.cppreference.com/w/cpp/language/operator_member_access

Assignment Operators[74]

- Operator = is often used for copying/moving (see next week)
- All assignment operators usually return *this

```
struct Int {
    int i;
    Foo& operator+=(const Foo& other) { i += other.i; return *this; }
};
Foo a{123};
a = Foo{456}; // a.i is now 456
a += Foo{1}; // a.i is now 457
```

[74] https://en.cppreference.com/w/cpp/language/operator_assignment

Conversion Operators[75]

- Conversion can be done using converting constructors (seen before)
- or *conversion operators*: `operator type ()`

- The `explicit` keyword can be used to prevent implicit conversions
- Explicit conversions are done with `static_cast`

```
struct Int {
    int i;
    operator int() const {
        return i;
    }
};
Int a{123};
int x = a; // OK, x is 123
```

```
struct Float {
    float f;
    explicit operator float() const {
        return f;
    }
};
Float b{1.0};
float y = b; // ERROR, implicit conversion
float y = static_cast<float>(b); // OK
```

[75] https://en.cppreference.com/w/cpp/language/cast_operator

`operator bool`

- `operator bool`: converts to `bool`
- Used to enable use of object in `if`, `while`, etc.
  - `if`, `while` etc. perform an *explicit* conversion

```
struct Ptr {
    void *p;
    explicit operator bool() const {
        return p; // pointers have an implicit conversion to bool
    }
};
Ptr p{nullptr};
if (p) {} // OK: explicit conversion
bool hasPtr = p; // ERROR: implicit conversion
```

Argument-Dependent Lookup[76]

- Overloaded operators are usually defined in the same namespace as the type of one of their arguments
- Regular unqualified lookup would not allow the following example to compile
- To fix this, unqualified names of functions are also looked up in the *namespaces of all arguments*
- This is called *Argument Dependent Lookup (ADL)*

```cpp
namespace A { class X {}; X operator+(const X&, const X&); }
int main() {
    A::X x, y;
    A::operator+(x, y); // OK
    x + y; // How to specify namespace here?
           // -> OK: ADL finds A::operator+()
    operator+(x, y); // OK for the same reason
}
```

[76] https://en.cppreference.com/w/cpp/language/adl

Enums[77]

- Typically used like integral types with a restricted range of values
- Also used to assign descriptive names instead of “magic” integer values
- Syntax: *enum-key name* { *enum-list* };
- *enum-key* can be `enum`, `enum class`, or `enum struct`
- Without explicit value, first element gets zero, other increment from previous

```
enum Color {
    Red, // Red == 0
    Blue, // Blue == 1
    Green, // Green == 2
    White = 10,
    Black, // Black == 11
    Transparent = White // Transparent == 10
};
```

[77] https://en.cppreference.com/w/cpp/language/enum

Using Enum Values

- Names from the enum list can be accessed with the scope resolution operator
- Enums can be converted to integers and vice versa with `static_cast`
- `enum` without `class`/`struct`: C-style enums
    - Names also introduced in the enclosing namespace
    - Can be converted implicitly `int`
- `enum class` and `enum struct` are equivalent
- Recommendation: Use `enum class` unless you have a reason not to

```
Color::Red; // Access with scope resolution operator
Blue; // Access from enclosing namespace
int i = Color::Green; // i == 2, implicit conversion
int j = static_cast<int>(Color::White); // j == 10
Color c = static_cast<Color>(11); // c == Color::Black
```

Type Aliases[78]

- Type names nested deeply in namespaces/classes can become very long

⇝ *Type alias*: `using` |*name*| `=` |*type*|`;`

- *name* is the name of the alias, *type* must be an existing type
- (C compatibility: equivalent to `typedef`, but prefer `using`)

```
namespace A::B::C { struct D { struct E {}; }; }
using E = A::B::C::D::E;
E e; // e has type A::B::C::D::E
struct MyContainer {
    using value_type = int;
};
MyContainer::value_type i = 123; // i is an int
```

[78] https://en.cppreference.com/w/cpp/language/type_alias

Classes and Conversions – Summary

- Classes are a sequence of their data members
- Classes can have member functions with implicit `this` pointer
- Member functions can be `const-qualified`
- Constructors are called for initializing objects
- Constructors and operators provide implicit/explicit conversions
- Class members can have different access control
- Access control can be circumvented by `friend` declarations
- Almost all operators can be overloaded with custom semantics
- Enums are, optionally scoped, integer types with descriptive value names

Classes and Conversions – Questions

- What is the difference between `class` and `struct`?
- When is padding required between fields?
- How can the size of a struct be reduced?
- What is the type of `this`? Is it always the same?
- Why do methods returning references typically have a non-const-qualified and a const-qualified overload? Which overload is taken in which cases?
- Why do references members have to be initialized in initializer lists?
- Why could massive operator overloading be problematic in large projects?
- How to access the raw integer value of `enum class` enumerators?