(5) C++ Abstractions

Nico Ludwig (@ersatzteilchen)

TOC

- Cited Literature:
  - Bjarne Stroustrup, The C++ Programming Language
  - John Lakos, Large-Scale C++ Software Design

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

Some words about Operator Overloading

- C++ supports the redefinition of present operators for UDT.
  - This means we can give already known operators a new functionality in our UDTs!
  - The set of operators we can use, is only depending on the types used in an expression.
  - We can then use the same operator for different types, therefor operator definition is basically operator overloading.
- C++'s aim is to mimic fundamental types as far as possible with UDTs:
  - The C++ Standard Template Library (STL) defines functions, which work for fundamental and user defined types.
  - To make these functions work with fundamental and user defined types, a "fundamental type mimicry" must be in place.
  - Therefor, the STL's functions only use operators to work with data internally.
  - Because operators can be overloaded for UDTs, operators play the role of a least common denomination of fund. types and UDTs.
- In comparison to Java overloaded operators play the role of interfaces in Java:
  - In Java, a UDT can optionally implement an interface to take part in algorithms of the JDK:
    - E.g. implementing the interface *Comparable* makes a UDT sortable with *java.util.Arrays.sort()*.
  - In C++, a UDT can optionally overload operators to take part in algorithms of the STL:
    - E.g. overloading the operator< makes a UDT sortable with *std::sort()* (declared in <algorithm>).
- On the following slides, we'll discuss the most essential operator overloads in C++.

- Java:
  - Java's ==-operator is automatically overloaded by the compiler to allow comparison of references.
  - Other interfaces:
    - Similar to C++'s cctors, Java's UDTs can implement the interface *Cloneable*. Copying is only rarely needed in Java, therefor *Cloneable* is used very rarely (and also criticized) in the wild.
    - Similar to C++'s dtors, Java's UDT can implement the interface *AutoCloseable*. – Explicit control over memory usage can then be taken over using try-with resource blocks, which slightly work like scope blocks in C++.

The Problem with shared Data – Copy Assignment

```
class Person { // (members hidden)
        char* name;
public:
        Person(const char* name) {
             if (name) {
                   this->name = new char[std::strlen(name) + 1];
                   std::strcpy(this->name, name);
             }
        }
        ~Person() {
             delete[] this->name;
        }
};
```

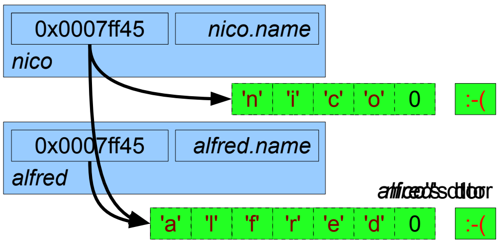
```
{
        Person nico("nico");
        Person alfred("alfred");
        nico = alfred; // operator= creates a copy.
} // Undefined behavior: crash!
```

```
{
        // Virtually following is executed:
        nico.name = alfred.name;
        // … for each field in Person!
}
```

- Let's revisit the type *Person*, which handles dynamic data.
  - Assigning a *Person*-object to another *Person* leads to undefined behavior in one dtor!
- The automatically created copy-operator= just copies all fields, not the referenced memory.
  - I.e. the copy-operator= of the UDT calls the copy-operator= for each field.
  - It results in *Person*-copies having copied pointers to the same location in memory (*name*).
  - Every *Person*-copy will try to delete (dtor) the same location in memory via its pointer-copy.

The automatically created Copy-Operator=

```
class Person { // (members hidden)
    char* name;
};
{
    Person nico("nico");
    Person alfred("alfred")
```
```
    nico = alfred;
```
```
}
```



- When *alfred* is assigned to *nico*:
  - The automatically generated copy-operator= does only assign *alfred*'s fields (i.e. *name*).
  - operator= doesn't manage the occupied memory in depth, it's a shallow assignment.
  - The automatically generated copy-operator= is public, also for classes!

- We have two *Person*s, both pointing to the same location in the freestore.
  - The memory referenced by *nico.name* before the assignment is lost, leading to a memory leak.
  - This leads to two *Person*s feeling responsible for *name*'s memory.
  - => In effect the same memory will be freed twice!
  - We hurt fundamental rules when dealing with dynamic memory!

Implementation of Copy-Operator= – A first Look

```
class Person { // (members hidden)
	char* name;
public:
	Person(const char* name) {
		// pass
	}
	// The overloaded operator=.
	Person& operator=(const Person& rhs) {
		if (this != &rhs) { // Self-assignment guard.
			delete[] this->name;
			this->name = new char[std::strlen(rhs.name) + 1];
			std::strcpy(this->name, rhs.name);
		}
		return *this;
	}
	~Person() {
		delete[] this->name;
	}
};
```

```
{
	Person nico("nico");
	Person alfred("alfred");
	nico = alfred;
} // Fine!
```

```
{
	// Virtually following is executed:
	nico.operator=(alfred);
}
```

Good to know
The parameter names lhs and rhs are sometimes used to denote parameters of an "operator-like" binary function. lhs: "left hand side", rhs: "right hand side".

- The solution: overload the operator= with correct copy semantics.
- We've to overload operator= to make a deep copy of the original to care for "stale" memory and to avoid memory leaks.

- As mentioned before, C++ is a language, in which the operator-support is based on functions. And here we're about to overload just these functions.

Implementation of Copy-Operator= in Detail

```
Person& Person::operator=(const Person& rhs) {
    if (this != &rhs) { // rhs is not identical to this.
        delete[] this->name; // (1)
        this->name = new char[std::strlen(rhs.name) + 1]; // (2)
        std::strcpy(this->name, rhs.name); // (2) continued
    }
    return *this;
}
```

- A correct implementation of copy-operator= should:
  - Accept a (preferably const) reference of the type to be assigned.
    - The parameter carries the object right from the assignment (*rhs*: "right hand side")
    - The parameter should be a const&, because we don't want to modify the *rhs* and to have it working for temporary objects!
  - Check the parameter against this (identity), so that no self-assignment can happen!
    - It could lead to freeing the memory in (1) that we want to copy in (2).
    - In short, this syntax should work:

```
nico = nico; // Self-assignment.
```

  - Free the assigned-to memory (1) and copy the assigning memory (2).
  - return a reference to this (Then no copy is created!), to make this syntax work:

```
Person joe("joe");
nico = alfred = joe; // Multiple assignment.
```

8

- An alternative to the shown implementation of the cctor and the copy assignment is to use the "copy and swap idiom".
- If an operator returns an object of same type, on which it was called, the operator (or in mathematical terms "the operation") is said to be "closed on that type".

The "Rule of Three"

- The automatically generated dtor, cctor and copy-operator= are often useless:
  - If a UDT should behave like a fundamental type and
  - if a UDT should follow value semantics (i.e. copy semantics).
  - And they are public also for classes!
- Therefor the "Rule of Three" claims that a UDT defining any of following "operators":
  - dtor,
  - cctor or
  - copy-operator=
  - should define all of them!

C++11 – explicit defaulting/deleting of special member functions

```
class NCPerson { // A non-copyable UDT Person.
public:
	NCPerson() = default;
	NCPerson(const NCPerson&) = delete;
	NCPerson& operator=(const NCPerson&) = delete;
};
```

- The "Rule of Three" completes our idea of RAII.
  - Dtor, cctor and copy-operator= are often called "The big Three".
- A UDT obeying the "Rule of Three" is often said to be in the "canonical form".

- Operators can be overloaded in order to mimic fundamental types, therefor ctors (esp. dctors and cctors), and dtors are also operators!

C++11: static Analysis of Type Traits

- C++11 provides means to analyze types during compile time.
  - These means where introduced to allow advanced and robust C++ meta programming.
  - But it can also be used learn something about C++ types.

C++11 – type traits

```
cout<<std::boolalpha<<std::is_copy_assignable<Person>()<<std::endl;
// >true

// Selection of type traits available in C++11:
std::is_array, std::is_class, std::is_copy_assignable, std::is_copy_constructible, std::is_enum,
std::is_floating_point, std::is_function, std::is_integral, std::is_lvalue_reference,
std::is_member_function_pointer, std::is_member_object_pointer, std::is_pointer,
std::is_rvalue_reference, std::is_union, std::is_void, std::is_arithmetic, std::is_compound,
std::is_fundamental, std::is_member_pointer, std::is_object, std::is_reference, std::is_scalar,
std::is_abstract, std::is_const, std::is_empty, std::is_literal_type, std::is_pod, std::is_polymorphic,
std::is_signed, std::is_standard_layout, std::is_trivial, std::is_trivially_copyable,
std::is_unsigned, std::is_volatile
```

- Meta programming is the "if-then-else of types".

When is no "Rule of Three" required to be implemented?

- If the "Big Three" are not explicitly coded, they'll be generated by the compiler:
  - The cctor, copy-operator= and the dtor of all the fields in the UDT will be called.
  - So, if all fields' types implement the "Big Three", nothing must be done in the UDT to be in the canonical form.
- Assume following UDT *MarriedCouple*:

```
class MarriedCouple {
        Person spouse1;
        Person spouse2;
public:
          MarriedCouple(const Person& spouse1, const Person& spouse2)
        : spouse1(spouse1), spouse2(spouse2) {}
};
```

  - The fields are of type *Person*.
  - The used UDT *Person* implements the "Big Three" and RAII.
  - *MarriedCouple* delegates its "Big Three" activities to its *Person* fields.
  - => No special member functions need to be implemented in *MarriedCouple*!

- If *std::string* was used in *Person* instead of char*, the "Big Three" wouldn't need to be implemented explicitly.

Lvalues and C++11 Rvalues

- Up to now we've dealt with two sorts of values:
  - Lvalues: "allowed left from assignment", have an address, controlled by programmers.
  - Non-lvalues: literals and temporarily created values (temps), have no known address.

- Non-lvalues can only be accepted by non-references and const&.
  - The problem: either is a (deep) copy created or a non-modifiable object (const&).

C++11 – rvalue declarator

```
void AcceptsPerson(Person&& person);
```

- C++11 closes the gap between lvalues and non-lvalues with rvalues.
  - Rvalues: modifiable references to non-lvalues controlled by compilers.
  - Rvalues allow binding non-lvalues to modifiable references implementing move semantics.
  - With move semantics literals and temps need not to be copied, but can be modified.
  - In C++11 we can control move-semantics additionally to copy-semantics.
  - Rvalues enable another new feature in C++: perfect forwarding.

- In C++11 the "Rule of Three" can be extended to the "Rule of Five" with move-semantics.

C++11: Move Semantics

- In C++ there are situations, in which deep copies are not really required.

```
AcceptsPerson(Person("dave")); // (1) Move constructor
nico = Person("joe"); // (2) Move assignment
```

- Neither the object *Person("joe")* nor *Person("dave")* is required after creation.
  - Their values are just stored in *AcceptsPerson()*'s parameter and in the variable *nico* respectively.
  - And then the rvalues *Person("joe")* and *Person("dave")* cease to exist.
- The idea is that the memory from the rvalues can be stolen instead of copied.
  - This is what we call move-semantics instead of copy-semantics.
  - In C++11 we can write code handling move construction (1) and move assignment (2).

C++11 – move assignment operator

```
Person& Person::operator=(Person&& rhs) {
    if (this != &rhs) {
        delete [] this->name;
        this->name = rhs.name; // Stealing!
        rhs.name = nullptr; // Ok on rvalue!
    }
    return *this;
}
```

C++11 – move constructor

```
Person::Person(Person&& original)
    : name(original.name) /* Stealing! */ {
    original.name = nullptr; // Ok on rvalue!
} /* or: */
// Implementation that forwards to move assignment:
Person(Person&& original) {
    *this = std::move(original); // Declared in <utility>.
}
```

13

- By default, most C++ compilers are able to optimize code in a way, so that this kind of superfluous copying is removed. The optimization of mtor and move assignment needs to be deactivated to make the explicitly implemented operators visible (gcc-option: *-fno-elide-constructors*).
- The function *std::move()* just converts its argument to a type to reach the correct overload for rvalues.

Essential Operators: Operator== and Operator!= – Part I

- What is equality? When do we understand objects to be equal?

```
Person nico1("nico");
Person nico2("nico");
```

  - Considering *nico1* and *nico2* as equal, we maybe assume <u>their fields have the same content</u>.

```
bool equalPersons = nico1 == nico2; // Won't compile for now!
```

  - Alas the operator== can't be used on *Person*s, because <u>this operator isn't defined yet</u>.

- What is identity?
  - The identity of objects can be <u>specifically answered for C/C++</u> ...
  - … <u>objects having the same address are identical</u>. Those objects <u>share the same location in memory</u>.

```
bool identicalPersons = &nico1 == &nico2; // Will almost always compile!
```

  - Identity-comparison is <u>intrinsic</u> in C/C++ as pointers are directly supported. <u>It is also called referential equality.</u>
  - We already checked for identity <u>to avoid self-assignment</u>:

```
if (this != &rhs) … // Self-assignment guard.
```

- Equality: <u>objects have the same content</u>. <u>It is also called structural equality.</u>
  - Once again in opposite to identity, where objects just have the same address.

- <u>What are the results in *equalPersons* and *identicalPersons*?</u>

Essential Operators: Operator== and Operator!= – Part II

```
class Person { // (members hidden)
public:
	bool operator==(const Person& rhs) const;
	bool operator!=(const Person& rhs) const;
};
```

- Good, let's implement operator==:

```
bool Person::operator==(const Person& rhs) const {
	if (this != &rhs) { // If rhs is not identical to this, compare the fields:
		return 0 == std::strcmp(this->name, rhs.name);
	}
	return true; // Self-comparison, both objects are identical: always true!
}
```

- And operator!= in terms of operator==:

```
bool Person::operator!=(const Person& rhs) const {
	return !this->operator==(rhs); // Just negate operator==.
}
```

- The signature of the equality operators shouldn't be a surprise:
  - Both are binary operators, the member functions accept the *rhs* and this is the "lhs".
  - They return a bool result and should be declared const.

Free Operators

- C++ allows operators to be overloaded as
  - member functions

```
bool Person::operator==(const Person& rhs) const { // Member function.
    if (this != &rhs) { // If rhs is not identical to this, compare the fields:
        return 0 == std::strcmp(this->name, rhs.name);
    }
    return true; // Self-comparison, both objects are identical: always true!
}
```

  - or as free functions.

```
bool operator==(const Person& lhs, const Person& rhs) { // Free function.
    if (&lhs != &rhs) { // If lhs is not identical to rhs, compare the fields:
        return 0 == std::strcmp(lhs.GetName(), rhs.GetName());
    }
    return true; // Self-comparison, both objects are identical: always true!
}
```

  - Mind that we can only access public members (*GetName()*) in the free operator==!

- Some operators can only be overloaded as member functions:
  - type conversion, operator=, operator(), operator[] and operator->.

Overload an Operator as free or Member Function?

- Often it depends on whether implicit conversion of the lhs is desired.
  - E.g. with having *Person*'s operator== as member function we'll have an asymmetry:

```
bool Person::operator==(const Person& rhs) const {
        // pass
}
```

```
Person nico("nico");
Person dave("dave");
bool result = nico == dave; // (1) Ok! Should be clear.
bool result2 = "nico" == nico; // (2) Invalid!
bool result3 = nico == "nico"; // (3) Ok!
bool result4 = "nico" == "nico"; // (4) Oops! Compares two const char*!
```

    - (2) An implicit conversion from const char* to *Person* works, but conversion of the lhs "nico" to "this" won't work!
    - (3) It works as the *rhs* const char* will be implicitly converted into *Person*. (2)/(3) are asymmetric!
    - (4) Just performs a pointer comparison; operators for fundamental types can't be "overridden".
    - (The compiler needed the free operator==(const char*, const *Person*&) to be present additionally.)
  - Then with having *Person*'s operator== as only one free function symmetry works:

```
bool operator==(const Person& lhs, const Person& rhs) {
        // pass
}
```

```
Person nico("nico");
Person dave("dave");
bool result = nico == dave; // (1) Ok! Should be clear.
bool result2 = "nico" == nico; // (2) Ok!
bool result3 = nico == "nico"; // (3) Ok!
bool result4 = "nico" == "nico"; // (4) Oops! Compares two const char*!
```

    - The implicit conversion makes (2) and (3) work symmetrically! (4) still doesn't call "our" operator==!

Friends

- Free functions and operator overloads can be granted access to private members.
  - This can be done by declaring them as friends of the UDT in the defining UDT:

```
class Person { // (members hidden)
	// The friendship declares that a free function with this signature
	// will get access to all members (incl. private members) of
	// Person. !!This is no declaration of a member function!!
	friend bool operator==(const Person& lhs, const Person& rhs);
};
```

  - With granted friendship, the free operator== can access *Person*'s private members:

```
// Free function as friend of Person:
bool operator==(const Person& lhs, const Person& rhs) {
	return (&lhs != &rhs)
		? 0 == std::strcmp(lhs.name, rhs.name) // private field name
		: true;
}
```

- When to use friends:
  - If access to "non-publicised" members is required.
  - If extra performance is awaited by accessing private members (often fields) directly.
  - Rule: avoid using friends! They're potentially dangerous as they break encapsulation.

Essential Operators: Operator<

- The implementation of the operator< for *Person* shouldn't be a surprise:

```
bool operator<(const Person& lhs, const Person& rhs) {
    return (&lhs != &rhs) ? 0 > std::strcmp(lhs.GetName(), rhs.GetName()) : false;
}
```

```
Person nico("nico"), dave("dave");
bool result = nico < dave;
// result = false
```

- The relational comparison operator< is very important in C++!
  - It can be used to implement all other relational comparison operators:

```
bool operator>(const Person& lhs, const Person& rhs) { return rhs < lhs; }
bool operator==(const Person& lhs, const Person& rhs) { return !(lhs < rhs) && !(rhs < lhs); }
bool operator!=(const Person& lhs, const Person& rhs) { return (lhs < rhs) || (rhs < lhs); }
bool operator>=(const Person& lhs, const Person& rhs) { return !(lhs < rhs); }
bool operator<=(const Person& lhs, const Person& rhs) { return !(rhs < lhs); }
```

  - Esp. the C++ Standard Template Library (STL) exploits the operator< like this.
    - E.g. the STL uses the operator< to implement sorting of elements!
- Expressing operator== in terms of operator< is called implementing equivalence by relative order.
  - I.e. we define equivalence not by equality, but in terms of the element-order in an ordered sequence.

- This example shows all the relational comparison operators available in C++. It's important to understand that these operators mustn't be overridden with pointer parameters. This is because we are not allowed to define binary operators only accepting pointers, as this would override the basic operators the language defines for pointers. It's also questionable (but allowed) to override binary operators in a way that only one of the parameters is a pointer type. – This would lead to a strange syntax when those operators are called.
- In C++ it is not required to overload all the relational operators separately as shown here. If only the operators < and == are defined for a type, based on these operators the other relational operators can be synthesized by the language. -> Just #include the h-file <utility> and add a using directive for the namespace *std::rel_ops*.

Essential Operators: Operator<<

- The operator<< is a binary operator, originally expressing left bit shift.
  - Traditionally it's overloaded to express "sending an object to an output stream" in C++.
    - E.g. can we send an object to the standard output-stream *std::cout* with its operator<<.
  - We can overload operator<< in "our" UDTs to exploit using streams for "our" UDTs.

- Overloading operator<< as member works, but the order of arguments is odd.

```
class Person { // (members hidden)
public:
        std::ostream& operator<<(std::ostream& out) const {
                return out<<GetName();
        }
};
```

```
Person nico("nico");
```

```
std::cout<<nico;   // Invalid! A Person object needs to be the lhs!
```

```
nico<<std::cout;   // Oops! Person's operator<< can't be used
// >nico           //  similar to fundamental types!
```

- Let's fix this and discuss this operator's signature.

Essential Operators: Operator<< – the correct Way

- In fact, the lhs of this operator<< needs to of type *std::ostream*; it can't be coded as member function!
  - Mind that in a member function the lhs would be this!

```
std::ostream& operator<<(std::ostream& out, const Person& rhs) {
    return out<<rhs.GetName(); // (1)
}
```

```
Person nico("nico");
Person dave("dave");
 // Ok! rhs as a Person object!
std::cout<<nico;
// (2) Chaining syntax:
std::cout<<" "<<dave<<" : "<<nico<<std::endl;
// >dave : nico
```

- So we've another reason to choose a free operator overload: order of arguments!

- The signature and implementation of operator<< for stream output of a UDT:
  - It accepts a non-const& to *std::ostream* as lhs (*out*) and a const& of the UDT as *rhs*.
    - The *lhs* is a non-const& because it is modified! The operator<< sends to, or writes to *out*!
    - As the *lhs* is of type *std::ostream*, it accepts *std::cout* as well *std::ofstreams* (substitution principle).
  - The implementation just calls operator<< on the passed *std::ostream* for its fields (1).
  - It returns the passed *std::ostream&* to allow chaining syntax (2)!

Operator Overloading – Tips

- Generally don't overload non-essential operators to have "aesthetic" code!
  - (Well, it's cool, because we kind of extend C++ by overloading present operators!)
  - In essence it only makes sense, if we define numeric types!
    - And the definition of numeric UDTs is not an every day's job.
    - Often it's an interesting school-exercise, but it's rarely done in the wild.
  - It's often done downright wrong – without analogy to fundamental types!
    - A little sneakier syntax isn't worth confusing consumers (arcane, not aesthetic) of such UDTs.
- Tips:
  - Operators modifying this/lhs should return a non-const& to this/lhs to allow chaining.
  - Some binary operators create and return a new instance of the UDT, no reference.
  - Keep operator and compound assignment (e.g. + and +=) consistent.
    - C++ won't infer compound assignment for us.
  - Never overload &&, || and , (comma/sequence operator) in C++!
  - Sometimes overloads of the operators++/--/</== are required for the STL to work.
  - Use free function operator overloads to exploit/control order of arguments or type conversion.
  - Most operators cannot be overloaded having default arguments, however, the function-call operator (operator()) can have any.

- The operators =, & and , (and the dctor and non-virtual dtor) have predefined meanings.

Thank you!