(4) C++ Abstractions

Nico Ludwig (@ersatzteilchen)

TOC

- Sources:
  - 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!

Cstrings revisited

- In the last lectures we talked about the encapsulation of concepts and RAII.
  - Now it's time to make use of RAII to make our programming tasks easier.
- E.g. we should apply RAII to encapsulate cstrings, thus enhancing:
  - creation,
  - assignment,
  - copying and
  - operations on cstrings
  - by encapsulating the tedious memory management around cstrings.
- The good news: we don't have to create a new UDT, we can use STL-strings!
  - STL-strings are represented by the C++ standard type *std::string*.
  - The type *std::string* is defined in <string>.
  - *std::string* provides important RAII features that make using *std::string* intuitive.

4

- Also the ugly 0-termination is to be encapsulated.
- The type *std::string* is a very good example of encapsulation. The design of *std::string* makes working with strings very comfortable, safe and productive, esp. because RAII has been implemented in *std::string*.
  - Many different libraries implemented "their" string types different from *std::string* (mutable string types, immutable string types, reference counted string types etc.), but we'll only use const char* or *std::string* in this course.

The UDT std::string

- To use STL-strings we have to #include <string> and keep on going with *std::string*:

```
std::string name = "Lola"; // Creation
std::string otherName = name; // Assignment/Copying
std::string subString = name.substr(2, 2); // Substring: a member function of std::string (result: "la")
// The memory of the used strings will be managed (e.g. also freed) by RAII.
```

- The type *std::string* can be used like a fundamental type, e.g. as return/parameter type:

```
// Using std::string as return and parameter type:
std::string AcceptsAndReturnsSTLString(std::string name) {
	std::cout<<"The passed name: "<<name<<std::endl;
	// Create and return an STL-string from cstring literal:
	return "Pamela";
}
```

```
// Create and accept STL-string from a cstring literal:
std::string result = AcceptsAndReturnsSTLString("Sandra");
// > The passed name: Sandra
// result = "Pamela"
```

- Hooray! – A simple-to-use string type in C++! But there is a performance problem:
  - Due to RAII, *std::strings* are copied when passed to and returned from functions.
  - It means that the encapsulated cstrings (char arrays) are allocated and freed multiply.
    - This is, e.g., done by ctors and dtors which manage RAII.
  - Can we improve this situation?

Avoiding Object Copies – Example: std::string

- Means to avoid multiple copying of *std::strings*:
  - Pass pointers to *std::strings* to functions (i.e. back to the idea "pass by reference").
  - Return pointers to *std::strings* created in the heap or freestore.
  - … let's rewrite *AcceptsAndReturnsSTLString()* accordingly:

```
// Using std::string* as return and parameter type:
std::string* AcceptsAndReturnsSTLString(std::string* name) {
	std::cout<<"The passed name: "<<*name<<std::endl;
	// Create std::string on freestore and return it.
	return new std::string("Pamela");
}
```

```
// Create std::string on the stack and pass a pointer to that
// string to the function:
std::string sandra = "Sandra";
std::string* result = AcceptsAndReturnsSTLString(&sandra);
// > The passed name: Sandra
// result = pointer to "Pamela"
delete result;
```

- The means work: neither *sandra* nor "Pamela" is copied, but created only once.

- But dealing with freestore and pointers to avoid copies is risky and cumbersome.
  - Therefor C++ introduced another means to avoid copies: references.

C++ References

- We can, e.g., use *std::string*-references as parameter types.
  - (For the time being, we'll only discuss parameters of reference type.)

```
// Using an std::string-reference as parameter type:
void AcceptsSTLString(std::string& name) {
	std::cout<<"The passed name: "<<name<<std::endl;
}
```

```
// Create std::string on stack and pass it to the function:
std::string pamela = "Pamela";
AcceptsSTLString(pamela); // Here: No extra syntax!
```

```
// Questionable! Field as C++ reference.
class Foo {
	int& refToInt;
public
	// The field refToInt needs to be initialized:
	Foo(int i) : refToInt(i){} // Initializer list needed!
};
```

```
// Questionable! Local variable as C++ reference.
int i = 23;
// The reference refToInt needs to be initialized:
int& refToInt = i;
```

- Syntactic peculiarities of C++ references:
  - A C++ reference has a syntax decoration of the referenced type with the &-symbol.
  - (+) References cannot be uninitialized and cannot be 0.
  - (+) When an argument is passed to a reference parameter, no extra syntax is involved.
    - Using reference parameters leads to more unobtrusive code than with pointer parameters.
  - (+) References can also be used as local variables and fields, but it leads to questionable code.
  - (-) References need to be initialized in opposite to pointers!
  - (-) References do have no notion of "nullity" like pointers (pointers can be 0).
  - (-) Functions differing only in the "reference-ness" of its parameters do not overload.

Definition
A C++ reference is a kind of pointer, that cannot be 0 or invalid/uninitialized.

Tracing Object Lifetime

```
class PersonLitmus {  // Shows, when an instance is
public:               // created and destroyed.
    PersonLitmus() {
        std::cout<<"Person created"<<std::endl;
    }
    ~PersonLitmus() {
        std::cout<<"Person destroyed"<<std::endl;
    }
};
```

- Let's reuse the type *PersonLitmus* with tracing messages.
  - Calling the function accepting *PersonLitmus* leads to anonymous copies:

```
void AcceptsPerson(PersonLitmus person) {
    // pass
}
```

```
PersonLitmus person;
// >Person created
AcceptsPerson(person);
// >Person destroyed (destroys anonymous copy)
// >Person destroyed (destroys person)
```

  - Calling the function accepting *PersonLimus*& avoids anonymous copies:

```
void AcceptsPersonByRef(PersonLitmus& person) {
    // pass
}
```

```
PersonLitmus person;
// >Person created
AcceptsPersonByRef(person);
// >Person destroyed (destroys person)
```

Anonymous Object Copies

- Let's review the example without reference parameters:

```
void AcceptsPerson(PersonLitmus person) {
        // pass
}
```

```
PersonLitmus person;
// >Person created
AcceptsPerson(person);
// >Person destroyed (destroys anonymous copy)
// >Person destroyed (destroys person)
```

- This example shows that more objects seem to be destroyed than created!
  - What the heck is going on here? The ctor was only called once, but the dtor twice!
- By default, C++ passes/returns objects to/from functions by value, creating copies.
- The answer for these unbalanced dtors: temporary copies of those objects are created.
- It's time to clarify the source of the anonymous copies: copy constructors (cctors).
  - Cctor are called when objects are getting copied!

Tracing Object Copies

```
class PersonLitmus {        // Shows when an instance is
public:                     // created, copied and destroyed.
    PersonLitmus() {
        std::cout<<"Person created ("<<this<<")"<<std::endl;
    }
    PersonLitmus(PersonLitmus& original) {
        std::cout<<"Person copied ("<<this<<")"<<std::endl;
    }
    ~PersonLitmus() {
        std::cout<<"Person destroyed ("<<this<<")"<<std::endl;
    }
};
```

```
void AcceptsPerson(PersonLitmus person) {
    // pass
}
```

```
PersonLitmus person;
// >Person created (0x7ffeefbff2b8)
AcceptsPerson(person);
// >Person copied (0x7ffeefbff2b0) (anon. copy created)
// >Person destroyed (0x7ffeefbff2b0) (destroys anon. copy)
// >Person destroyed (0x7ffeefbff2b8) (destroys person)
```

- Implementing the copy constructor with tracing makes the creation of copies visible.
  - If not explicitly programmed, a public cctor will be generated automatically (like dctors/dtors are).
- Syntactic peculiarities of cctors:
  - It has exactly one (non-defaulted) parameter: a reference of the surrounding type.
  - The cctor obeys to the same syntactic rules like other ctors do.
  - A type's cctor will be automatically called, when the type is passed/returned by value.

- Why needs a cctor to accept a reference of the surrounding type?
  - If the cctor accepted a non-reference parameter of the surrounding type, the cctor would call itself recursively up to infinity!
- Cctors' parameters are allowed to be annotated with all kinds of cv-qualified references, they also overload (That should not be done on purpose!). – The rules for matching overloads are complex and will not be discussed here.
- Cctors can have more than the obligatory parameter, but the remaining parameters need to be notated with default arguments.

The Problem with shared Data – Copying

```
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;
        }
};
```

```
void AcceptsPerson(Person person) {
        // pass
}
```

```
Person nico("nico");
AcceptsPerson(nico);
// Undefined behavior: crash!
```

```
// Virtually something like this is executed:
Person nico("nico");
Person tmp;
tmp.name(nico.name); // Copies the pointer.
AcceptsPerson(tmp);
```

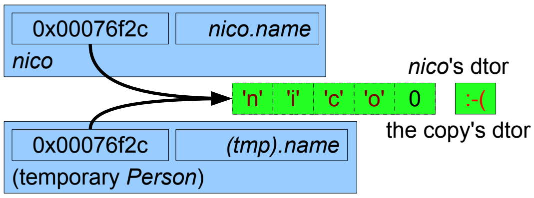
- Let's revisit the type *Person* that handles dynamic data.
  - Passing an object of type *Person* by value leads to undefined behavior in the dtor!
- The automatically created cctor just copies all fields, not the referenced memory.
  - It results in *Person*-copies having copied pointers to the same location in memory (*name*).
  - Every *Person*-copy will try to delete (dtor) the same location in memory via its pointer-copy.

- The syntax *tmp.name(nico.name)* is not legal, a cctor can't be called like so. – But this is what's happing underneath basically.

The automatically created Copy Constructor

```
class Person { // (members hidden)
    char* name;
};
```

```
{
    Person nico("nico");
    AcceptsPerson(nico);
} // Undefined behavior when nico's scope ends!
```



- When *nico* is passed to *AcceptsPerson()* a copy of *nico* is created.
  - The automatically generated cctor does only copy *nico*'s fields (i.e. *name*).
  - The cctor doesn't copy the occupied memory in depth, we call this a shallow copy.
  - The automatically generated cctor is public also for classes!

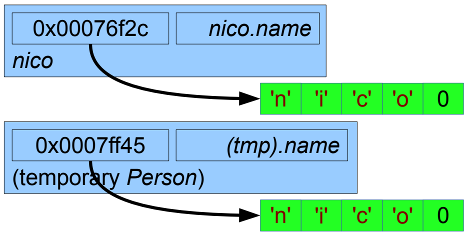
- We've two *Persons*' *name*-fields both pointing to the same location in the freestore.
  - This leads to dtors of two *Persons* feeling responsible for *name*'s memory.
  - => In effect the same memory will be freed twice!
  - We hurt a fundamental rule when dealing w/ dynamic memory: We should not free dynamically created content more than once!

The Copy Constructor

```
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(Person& original) { // The cctor.
		name = new char[std::strlen(original.name) + 1];
		std::strcpy(name, original.name);
	}
	~Person() {
		delete[] this->name;
	}
};
```

```
void AcceptsPerson(Person person) {
	// pass
}
```

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



- The solution: we've to implement the cctor explicitly!
  - We have to implement the cctor to make a deep copy of the original.
    - The cctor just accepts the original object and makes a deep copy.
  - *std::string* provides a cctor.

Implicit and Explicit Conversion Constructors

- In fact we have already overloaded an operator in the UDT *Person*!
  - Every single-parameter ctor is also an implicit conversion ctor.
  - E.g. a const char* passed to *AcceptsPerson()* will be implicitly converted to a *Person*.
  - Also *std::string* has such an implicit conversion ctor: *string::string(const char*)*.

```
class Person { // (members hidden)
public:
        Person(const char* name) {
            // pass
        }
};
```

```
void AcceptsPerson(Person person) {
        // pass
}
```

```
AcceptsPerson("nico"); // Ok!
```

- Sometimes implicit conversion is not desired for single-parameter ctors.
  - (E.g. to avoid "surprises" with overloaded functions.)
  - Therefor C++ allows conversion ctors to be marked as explicit conversion ctors.

```
class Person { // (members hidden)
public:
        explicit Person(const char* name) {
            // pass
        }
};
```

```
void AcceptsPerson(Person person) {
        // pass
}
```

```
AcceptsPerson("nico"); // Invalid! No implicit conversion!
```

```
AcceptsPerson(Person("nico")); // Ok! Explicit conversion.
```


- *HandlePerson*(*Person*) and *HandlePerson*(const *Person*&) would not be called, if *HandlePerson(const char*)* existed and *HandlePerson("nico")* be called. The best match is taken, implicit conversion and temporary copies are avoided at overload resolution.

Problems with Reference Parameters

- After the discussion concerning costly copying, let's use references for all UDTs to avoid (deep) copying in future! – But there are problems:
  - 1. With functions accepting references we could modify the original argument!
    - Very often this happens accidentally, here it is shown in a more radical example:

```cpp
void PrintToConsole(std::string& text) {
	std::cout<<text<<std::endl;
	// Oups! Modified the parameter and the argument!
	text = "Angela";
}
```

```cpp
std::string pamela = "Pamela";
PrintToConsole(pamela);
// >Pamela
std::cout<<pamela<<std::endl;
// >Angela (Oups! The content of pamela has been modified!)
```

  - 2. Functions accepting references can't cope with objects created by implicit conversions.
    - E.g. a const char* can't be implicitly converted into an *std::string* and passed to an *std::string&*:

```cpp
void PrintToConsole(std::string& text) {
	std::cout<<text<<std::endl;
}
```

```cpp
// Invalid! Literal const char* can't be passed!
PrintToConsole("Pamela");
```

    - This yields a compiler message like "Non l-value can't be bound to non-const reference."

- An object created by implicit conversion is a temporary object, and as such it can't be an lvalue, therefor this message happens to appear.

Const References

- The discussed problems can be solved with const references (const&).

```
void PrintToConsole(const std::string& text) {
        std::cout<<text<<std::endl;
        // Invalid! Const parameter.
        text = "Angela";
}
```

```
std::string pamela = "Pamela";
PrintToConsole(pamela);
// >Pamela
```

- const& parameters cannot be modified (accidentally), e.g. assigned to.
  - const *std::string&* are esp. important as *std::string* is a mutable string type.
  - The same is valid for const pointers (e.g. also as parameters).

- const& parameters can accept temporary anonymous conversion copies.
  - A const char* can be implicitly converted into an *std::string* and passed to a const *std::string&*:

```
void PrintToConsole(const std::string& text) {
        std::cout<<text<<std::endl;
}
```

```
// Fine! Literal const char* can be passed now!
PrintToConsole("Pamela");
```

16

- This is a very important fact: const char* is non-modifiable, but the STL type *std::string* is mutable! With const *std::string*s, (and const& thereof) we can get const char*'s safety aspect back.
- Strings in Java and .NET are not mutable. On that platforms string operations create new strings instead of modifying the string on which they have been called.

The need for const Member Functions

- Ok! Then let's use const& for all UDTs! – But there are still problems:
  - We can't call all the member functions via a const&.

```
class Date { // (members hidden)
        int month;
public:
        int GetMonth() {
            return month;
        }
};
```

```
void PrintMonth(const Date& date) {
        // Invalid! Can't call non-const member function on const&.
        std::cout<<date.GetMonth()<<std::endl;
}
```

```
void PrintMonth(const Date* date) {
        // Invalid! Can't call non-const member function on const pointer.
        std::cout<<date->GetMonth()<<std::endl;
}
```

- C++ assumes that all member functions potentially modify the target object.
  - Calling member functions on const& and const pointers is generally not allowed.
  - Compilers can't predict, whether called member functions modify the object.
- To solve this problem C++ introduces so called const member functions.
  - Only const member functions can be called on const objects.

Implementing const Member Functions

- To make a member function const: just add the const suffix to declaration and definition.

```
class Date { // (members hidden)
	int month;
public:
	int GetMonth() const {
		return month;
	}
};
```

```
void PrintMonth(const Date& date) {
	// Fine! Date::GetMonth() is a const member function.
	std::cout<<date.GetMonth()<<std::endl;
}
```

```
void PrintMonth(const Date* date) {
	// Fine! Date::GetMonth() is a const member function.
	std::cout<<date->GetMonth()<<std::endl;
}
```

- In the implementation of a const member function
  - fields of the enclosing UDT/instance can only be read,
  - only static member functions and other const member functions of the enclosing UDT/instance can be called.
  - The const-ness of const member functions is kind of "closed".
  - *std::string* provides a set of const member functions.
- Member functions and const/volatile qualifiers (cv-qualifiers):
  - We can have a const and a non-const overload of a member function in C++.
  - C++ also allows the definition of volatile and const volatile member functions.
    - Non-const, const, volatile and const volatile do overload!

- What is "volatile"?
  - If an object is declared volatile, the C/C++ compiler assumes that its value could change anytime by any thread of execution. – Accessing a volatile variable is interpreted as direct memory read/write without caching.
- The cv-qualification needs to be repeated on a non-inline definition, because cv-qualifiers overload.
- There can be no const static member functions.

C++ References summarized: Our Rules for References

- C++ references have been introduced to overload operators, because:
  - they provide an unobtrusive syntax to make using operators intuitively,
  - they prevent call by value and
  - const& allow passing temporary objects, so its also good for non-operator functions.

- Important features of C++ references:
  - They're similar to pointers as call by reference, aliasing and data sharing is concerned.
  - References can't be uninitialized and they have no notion of "nullity".

- Some widely used industry standards that we're going to adopt as rules:
  - (+) Primary use const& for UDT parameters, and avoid passing UDTs by value!
    - UDTs should be passed by const&, if they are not already being passed as const pointer.
  - (+) Pass fundamental types only by value or pointer, the compiler cares for optimization.
  - (+) If we need to modify passed objects we should use non-const pointers.
  - (-) Don't use references to replace pointers only for syntax purposes in order to modify the passed objects!

- If we see the discussed rules hurt, then this is what we call a "code smell". If the code is under our control, we should refactor it to obey these rules and prove the functionality with unit tests.

Circumventing const: mutable Fields

- We can't modify the target object (i.e. this) in const member functions.
  - We can not write a const *Date::SetMonth()* member function like so:

```
class Date { // (members hidden)
        int month;
public:
        void SetMonth(int month) const {
            this->month = month; // Invalid! this->month is readonly
        }                         // in a const member function.
};
```

- Often real life objects need to differentiate logical from physical const-ness.
  - I.e. an object presents const member functions that need to write some fields.
  - C++ provides a backdoor: mutable fields can be written in const member functions.

```
class Date { // (members hidden)
        int month;
        mutable int monthAccessed;
public:
        int GetMonth() const {
            ++monthAccessed;    // Fine! Mutable fields can be written
            return month;       // in const member functions.
        }
};
```

Cheating const: Casting const-ness away

- After discussing how to avoid copies, implicit conversion and *std::string*, we'd better use const *std::string&* as function parameter type always.
  - But this also means that passed *std::string*s can't be modified!

```
void Clear(const std::string& name) {
	name.erase(); // Invalid! Can't call non-const member function on name.
} // That makes sense! std::string::erase() can't be a const member function!
```

- There is a way to call non-const member functions on const objects!
  - const-ness can be casted away in C++ with the const_cast operator:

```
void ClearConst(const std::string& name) {
	std::string& nonConstName = const_cast<std::string&>(name);
	nonConstName.erase(); // Ok! nonConstName is not const!
}
```

- Casting const-ness away is dangerous! – It's undefined with temporary objects.

```
std::string nico("nico")
ClearConst(nico); // Ok!
std::cout<<nico<<std::endl;
// >    // (empty)
```

```
ClearConst("nico");     // Undefined behavior! The literal "nico"
                        // is really const and can't be erased!
std::cout<<nico<<std::endl;
// >    // (empty)
```

  - Let's never use const_cast, unless we've a very good reason to use it!

"Const-incorrectness"

- All fields of a const object are also const, we can only call const member functions on it.

- Assume the UDT *Person* and the const member function *GetName()* in (1):

```
class Person { // (1) (members hidden)
public:
        char* name; // Only for demo!
        char* GetName() const {
             return this->name;
        }
};
```

```
class Person { // (2) (members hidden)
public:
        char* const name;
        char* GetName() const {
             return this->name;
        }
};
```

- When we define a const instance of *Person*, the UDT virtually changes to (2).

```
// We already know this fact:
const Person nico("nico");
// This is not allowed:
nico.name = 0; // Invalid!
// The field nico.name is a const pointer!
```

```
const Person nico("nico"); // But here the surprises:
// These operations are allowed(!):
std::strcpy(nico.name, "joe");
// The memory to which nico.name points to is not const!
std::strcpy(nico.GetName(), "jim");
// The memory to which nico.name points to is not const!
// - GetName() just returns a pointer.
```

- Was all our work for const-ness for good? – Well, our UDT isn't yet const-correct!

22

- Another way to understand const correctness: const member functions need to be balanced to other const member functions and their return types. Finally, const correct types allow the creation of immutable types in C++. (The support for immutable types is not so good in Java and C#.)

Const-correctness

- The just encountered problem is that const-ness is not deep enough!
  - The const-ness of an object only affects the fields, not the memory the fields refer to.
    - This problem is relevant for pointer-fields as well as for reference-fields.
  - const member functions don't shield us from modifiable referred memory!

- To make *Person*/*GetName()* const-correct, we'll make the field *name* private and fix *GetName()* to return a const char* instead of a char*.
  - 1. The field *name* can't be accessed from "outside".
  - 2. *GetName()* returns a const char*, it doesn't allow writing memory referenced by *name*.

```
class Person { // (members hidden)
	char* name;
public:
	const char* GetName() const {
		return this->name;
	}
};
```

```
const Person nico("nico"); // Now Person is const correct.
```

```
std::strcpy(nico.name, "joe"); // Invalid! name is private!
```

```
std::strcpy(nico.GetName(), "jim"); // Invalid! GetName()
// returns a const char* that doesn't allow to write the
// referred memory.
```

- const correct: Generally return const pointers/references from const member function.

Thank you!