(2) C++ Abstractions

Nico Ludwig (@ersatzteilchen)

TOC

- Cited Literatur:
  - Bruce Eckel, Thinking in C++ Vol I
  - Bjarne Stroustrup, The C++ Programming Language

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!

Limits of UDTs/Structs as "Records"

- Another way to understand UDTs: a try to simulate the reality.
  - The UDT *Date* is a working, concrete and every-day concept of our reality!

- But record-oriented programming has still some limitations:
  - The "belonging together" of functions and UDTs is not obvious.
  - The UDT instance that is "passed around" is not encapsulated.
    - Access and manipulation of the object is possible outside of "its" functions.
    - We could set the *day* of a *Date* directly to the value 200, breaking the concept...
  - There is a separation of data (UDT instances) and functions (operations).

- Frankly, some concepts can be retained as they proved well in record-orientation:
  - Instances/objects of UDTs are needed to simulate "things" existent in the real world.
  - Functions are needed to simulate operations with objects.

- We should combine UDTs (data) and functions (operations) in a better way!

- Up to now we've only discussed UDTs composed of other types still making up relatively simple types, e.g. *Date*, *Coordinate* and *Fraction*. But this is only the base for more complex and mightier types in C++ we're going to discuss in this lecture.

Concepts of Object Orientation

- Effectively, the combination of self contained data and behavior is our aim.
- In sum, following concepts are required to implement this aim:
  - 1. Abstraction by combining data and functions into a type.
  - 2. Encapsulation to protect data from unwanted access and modification:
    - The *day*-part of a *Date* instance should not be modifiable from "outside".
  - 3. The whole – part (aggregation or composition) association:
    - We can say "A car object has an engine object.".
  - 4. The specialization – generalization association:
    - We can say "three cars drive in front of me", rather than saying that there "drives a van, a bus and a sedan in front of me". The generalization is possible, because e.g. a van is a car.
- "Object-orientation" (oo) is the umbrella term for these concepts.
  - Oo languages provide features that allow expressing these concepts.
  - In this lecture we're going to understand abstraction and encapsulation.

Abstraction of Data and Behavior in UDTs

- Let's assume following struct *Date* and its belonging to function *PrintDate()*:
  - Record-types that only contain fields are often called Plain Old Datatypes (PODs).

```
struct Date { // A POD.
	int day;
	int month;
	int year;
};
```

```
// The definition of Date's belonging to function PrintDate():
void PrintDate(Date date) {
	std::cout<<date.day<<"."<<date.month<<"."<<date.year<<std::endl;
}
```

- In C++ we can put the belonging to functions into a struct definition:
  - *PrintDate()* is now a member function of *Date*.
  - *PrintDate()* can directly access a *Date*-object's data.
    - So the formally awaited parameter "*Date date*" is nolonger required.
    - *Date*'s data and the function *PrintDate()* are now combined into one UDT.
  - At the function definition, *PrintDate()* needs to be defined as *Date::PrintDate()*.
  - Like free functions member functions can have overloads.

```
struct Date { // Definition of the UDT "Date" composed of
	int day; // data (fields) and functions.
	int month;
	int year;
	void PrintDate(); // Member function declaration.
};
void Date::PrintDate() { // Function definition.
	std::cout<<day<<"."<<month<<"."<<year<<std::endl;
}
```

Hint
Upcoming examples assume, that the structs and their member function definitions, all free functions and esp. *main()* reside in the same c-file. – We will revisit the aspect of multi file projects in a future lecture.

6

- Member functions are somehow the "opposite of free functions".

Abstracted Types – Definition

- With the combination of fields and functions into a type we have an abstracted type.
  - Data (fields/record-type/POD) + member functions = abstracted type

- Definition of an abstracted type:

```
struct Date { // An abstracted type.
	int day;
	int month;
	int year;
	void Print();
};
```

```
// Member function definition.
void Date::Print() {
	std::cout<<day<<"."<<month<<"."<<year<<std::endl;
}
```

  - The UDT (struct) should be defined separately.
    - It defines all the fields and declares, or mentions the member functions.
  - All the member functions should be defined separately as well.
    - It makes sense to rename *PrintDate()* to *Print()*, because *Print()* has now a clear context.
  - All fields and member functions of a UDT are summarized as members of the UDT.

- A UDT can also have other UDT definitions as members, so called inner types.

Abstracted Types – Calling Member Functions

- We can use instances of the abstracted type *Date* like this:
  - With the dot-notation the fields of a *Date* instance can be accessed.
  - With the dot-notation *Date*'s member function *Print()* can be called as well.

```
Date myDate;
myDate.day = 17; // The individual fields can be accessed w/ the dot-notation.
myDate.month = 10;
myDate.year = 2012;
myDate.Print(); // The member functions can be called w/ the dot-notation as well.
// >17.10.2012
```

- All the members of a *Date* pointer can be accessed with arrow-notation:

```
Date* pointerToDate = &myDate;
pointerToDate->day = 18;  // The individual fields can be accessed w/ the arrow-notation.
pointerToDate->Print(); // The member function can be called w/ the arrow-notation as well.
// >18.10.2012
```

Problems with UDT Initialization

- We should refine the design of *Date*. Some serious problems remained!
  - We could forget to initialize a *Date* instance with very unpleasant results:

```
// Create a Date instance and assign _none_ of its fields.
Date myDate;
myDate.Print();
// >39789213.-2198627.-8231235 Ouch!
```

  - We could initialize a *Date* instance incompletely also with very unpleasant results:

```
// Create a Date instance and assign values to _some_ of its fields:
Date myDate;
myDate.day = 17;
myDate.month = 10;
myDate.Print();
// >17.10.-8231235 Ouch!
```

  - We could initialize a *Date* instance more than once, this also has with very unpleasant results:

```
// Create a Date instance and assign values its fields for two times:
Date myDate;
myDate.day = 17, myDate.month = 10, myDate.year = 2012;
myDate.day = 20, myDate.month = 5, myDate.year = 2011;
myDate.Print();
// >20.5.2011 Ouch!
```


- Notice that the word "design" was used. Mind that we try to simulate the reality, and "simulating the reality" or "simulating the nature" is another definition of the term "art". Oo programming has many parallels to art, esp. do oo-programmers have to work creatively.

Improving UDT Initialization with Constructors

- We can fix all three problems with a so called constructor (ctor).
  - Here the updated definition of *Date* with a ctor:

```
struct Date {
        int day;
        int month;
        int year;
        Date(int d, int m, int y); // constructor
        void Print();
};
```

```
// The ctor assigns the fields of a new Date instance
// for us:
Date::Date(int d, int m, int y) {
        day = d;
        month = m;
        year = y;
}
```

  - The definition of the ctor should go separately as for all other member functions.

- Facts about ctors:
  - A ctor has the name of the enclosing UDT.
  - A ctor is a member function that initializes an instance of a UDT.
  - A ctor often has parameters to accept values for the initialization of the instance.
    - *Date*'s ctor accepts initial values for all of its fields in the parameters *d*, *m* and *y*.
  - A ctor doesn't return a value and has no declared return type. Not even void!

- Why were the parameters named *d*, *m* and *y* and not *day*, *month* and *year*?
- Like other (member) functions, ctors can have overloads and default arguments.

Calling Constructors

- The definition of ctors is one thing, but their usage is far more interesting!

```
// Create a Date instance with the ctor and pass values to initialize its fields:
Date myDate(17, 10, 2012); // The ctor performs the assignment of the fields!
myDate.Print();
// >17.10.2012
```

- The syntax of calling a ctor is like calling a function while creating an instance.
  - Indeed ctors are functions. – Only the definition and usage is somewhat special.
  - If we define any ctor in a UDT, instances of that UDT cannot be created with initializers.

```
// Initializers can't be used, if a UDT has at least one ctor:
Date aDate = {17, 10, 2012}; // Invalid in C++03!
// (In C++11 this is handled with uniformed initializers.)
```

- Due to the syntax of the ctor call:
  - 1. There is no way to forget to call the ctor!

```
Date myDate; // Invalid! Doesn't call the ctor!
```

  - 2. There is no way to call a ctor and miss any of the initialization values!
    - You have to pass arguments to satisfy all of the ctor's parameters!
  - 3. There is no way to call a ctor more than once on the same instance!
    - Multiple initialization is not possible.

The default Constructor – Definition and Usage

- After we have defined our handy ctor, we have to use it always for initialization:

```
Date myDate; // Invalid! We have to call a ctor!
```

```
Date anotherDate(17, 10, 2012); // Ok! Calls the ctor.
```

- Additionally, we should also define a default constructor (dctor).

```
struct Date { // (members hidden)
	Date();  // dctor
	Date(int d, int m, int y); // ctor
};
```

```
Date::Date() {  // This dctor assigns the
	day = 1;        // fields of a new Date
	month = 1;      // instance to meaningful
	year = 1978;  // default values.
}
```

C++11 – in class initialization of fields

```
struct Date {
	int day = 1;
	int month = 1;
};
```

  - A dctor initializes a UDT with a default state, i.e. with default values for a UDT's fields.
  - Usually this means that all fields are initialized with values that are meaningful defaults.

- So as ctors are functions, a dctor is the parameterless overload of the ctor.
  - Let's use both *Date* ctors to create two instances of *Date*:

```
Date myDate; // Ok! Calls the dctor.
myDate.Print();
// >1.1.1978
```

```
Date anotherDate(17, 10, 2012); // Call the other overloaded ctor.
anotherDate.Print();
// >17.10.2012
```

Good to know
The special term "default constructor" is common sense in many languages and frameworks, sometimes it is called "parameterless constructor". Special terms like "standard constructor" or "common constructor" (German: "allgemeiner Konstruktor") do simply not exist officially. The leading sources for technical terms are specs and compiler messages, neither professors nor teachers or books.

- Dctors can have parameters, but those need to be declared with default arguments.

The default Constructor – Consequences

- If we don't provide any ctor, a dctor will be implicitly created by the compiler.
  - This created dctor does calls the dctors of the fields.
  - Dctors of fundamental types don't initialize to meaningful defaults: the fields' values are undefined!
  - So this dctor is implicitly created if not provided, therefor it is called default ctor!
  - But the compiler generated dctor is almost never useful!
    - Therefor the dctor of a UDT should be explicitly defined by the programmer!
- If we do provide any ctor that is no dctor, no dctor will be created by the compiler.
  - Because of this fact we had to implement a dctor in the type *Date*.
- To allow creation of "bare" arrays, a UDT must provide a dctor. The dctor will then initialize all elements of such an array.
  - "Bare" arrays of UDTs aren't really bare, but default initialized! (The default initialization of fund. types results in undefined values!)

```
Date dates[10]; // Will create an array of ten Date objects. I.e. the dctor will be called for ten times!
```

Temporary Instances and Constructors of fundamental Types

- With ctors we can pass temporary instances of UDTs to functions.
  - Temporary instances are instances that are not explicitly assigned to variables.
  - Let's revisit the free function *PrintDate()*:

```
void PrintDate(Date date) {
    std::cout<<date.day<<"."<<date.month<<"."<<date.year<<std::endl;
}
```

  - We can pass temporary *Date* instances to *PrintDate()* like so:

```
PrintDate(Date(17, 10, 1978)); // A new Date instance is created "on the fly".
// >17.10.1978
PrintDate(Date()); // A new Date instance is created "on the fly" with the dctor.
// >1.1.1978
```

- We can also create (temporary) instances of fundamental types with the ctor syntax:

```
char ch = std::toupper(int('c')); // Passes a temporary int to std::toupper().
int i(42); // Calls a ctor of int to initialize i with the value 42.
int e = int(); // Calls the type-default ctor of int to initialize e w/ the type-default value of int (0).
int k; // Ouch! int's dctor will initialize k to an undefined value!
int j(); // Ouch! This statement doesn't create an int! - It declares a function j returning an int.
```

  - This style of creating temporary instances is also a kind of conversion (cast).
  - Other perspectives of creating temporary instances: function cast or cast ctor.

- The dctor of fundamental type initializes to undefined values, but the <u>type-default constructor</u> initializes them to the values 0 or false.
- For int('c') we can read "int of 'c'". – It looks and sounds very like a conversion from char to int is happening here, so when a ctor is called for a conversion it is sometimes called cast-constructor. Another perspective is to understand the ctor call as a function call and then the conversion can also be called function-cast. Finally we have three syntaxes for conversions in C++: C-casts, functional-casts (C++ only) and C++-casts.

Implementation of Constructors – Initializer Lists

- Alternatively we can define a ctor with an initializer list to initialize fields.

```
// This dctor initializes the fields of a Date with an initializer list directly to
// explicit default values. No assignment is done, the ctor's body is empty!
Date::Date() : day(1), month(1), year(1978) { /* Empty! */ }
```

```
Date().Print(); // Calling Print() on a temporary object.
// >1.1.1978
```

  - Initializers in the list can be left empty, then the fields' values have type-default values.

```
// A dctor with an initializer list for type-default values for the fields.
Date::Date() : day(), month(), year() {} // For three ints the dctor is called.
```

```
Date().Print();
// >0.0.0
```

- Initializer lists are sometimes required:
  - for const fields, as they can't be assigned, for fields whose type has no dctor,
  - for references, because they need to be initialized and to call ctors of base types.
- The order, in which the fields get initialized depends on the order in the UDT!
  - I.e. it doesn't depend on the order in the initializer list!

C++11 – uniformed initializer lists in ctors:

```
struct Person {
        Date birthday;
        // Initialize birthday w/ a unified initialization list:
        Person() : birthday{23, 11, 1972} { /* pass */ }
};
```

C++11 – delegation:

```
struct Date { // (members hidden)
      Date(int day, int month, int year) { /* pass */ }
      // Calls the ctor Date(int, int, int):
      Date() : Date(1, 1, 1978) { /* pass */ }
};
```

Implementation of Constructors – the this-Pointer

- Each (instance-)member function can access the current instance's members.
  - This can be done implicitly, simply by "using" fields and other member functions:

```
void Date::Print() {
    std::cout<<day<<"."<<month<<"."<<year<<std::endl;
}
```

  - Or explicitly by accessing the current instance via the this-pointer:

```
void Date::Print() {
    std::cout<<this->day<<"."<<this->month<<"."<<this->year<<std::endl;
}
```

- The this-pointer is required to, e.g., distinguish parameters from fields:

```
struct Date { // (members hidden)
    int day;
    int month;
    int year;
    Date(int day, int month, int year);
};
```

```
// The ctor assigns the fields via the this-pointer:
Date::Date(int day, int month, int year) {
    this->day = day;
    this->month = month;
    this->year = year;
}
```

  - It's also required, if the current instance needs evaluation (e.g. in the yet to be discussed copy constructor).

- Mind how the arrow (->) notation comes in handy!

16

- The this pointer is also useful to trigger code completion on the members of the current instance in the IDE.

Unrestricted Access – A Problem rises!

- Let's assume the already defined UDT *Date* will be used like this:

```
Date myDate(17, 10, 2012); // Ok, construct the Date.
myDate.month = 14; // Oups! Quattrodecember??
myDate.Print();
// >17.14.2012
```

- What have we done?
  - We can freely access and modify all fields of the struct *Date*, so far so good...
  - We can also set all the fields to invalid values as far as *Date*'s concept is concerned.

- How can we fix that?
  - We have to encapsulate all fields of the UDT (e.g. *month*)! This requires
  - 1. Each field can only be read and modified with member functions, incl. the ctors.
    - 1.1 E.g. a set-function should check the value to be set! – We can force the validity of a *Date*!
  - 2. Direct access to the fields has to be restricted for the users of *Date*.
    - 2.1 *Date*'s member functions have to be publicly accessible.
    - 2.2 *Date*'s fields have to be privately accessible.

Restricting Access to Members (esp. Fields)

```
struct Date { // (members hidden)
private:
	int month;
public:
	Date(int day, int month, int year);
	int GetMonth();
	void SetMonth(int month);
};
```

```
Date myDate(17, 10, 2012); // Construct the Date.
myDate.SetMonth(14); // Try to set quattrodecember.
myDate.Print(); // myDate remains 17.10.2012!
// >17.10.2012
```

```
// Accessor member function of month.
int Date::GetMonth() {
	return month;
}
// Manipulator member function of month.
void Date::SetMonth(int month) {
	if (1<= month && month <= 12) {
		this->month = month;
	}
}
```

- What have we done?
  - 1. The UDT *Date* has been separated into two parts, labeled as private and public.
  - 2. The member functions *GetMonth()*/*SetMonth()* handle/care for the field *month*.

```
std::cout<<myDate.month<<std::endl; // Invalid! Can't access private field month.
std::cout<<myDate.GetMonth()<<std::endl;  // Fine, uses the Get-function!
// >12
```

- Effectively
  - the field *month* can't be directly accessed/modified, but via *GetMonth()*/*SetMonth()*,
  - esp. *SetMonth()* checks the validity of the *month*(-parameter) to be set.

- Some slides before we defined PODs as UDTs that only contain fields. There are different opinions on when being a POD ends. One way to understand PODs is having only public fields. Another way to understand them is a UDT having only fields with the same access modifier. The later definition requires UDTs having a clearly defined memory layout, and the memory layout (i.e. fields have increasing addresses) can be different depending on whether (different) access modifiers are used in a UDT.
- Concerning the access modifiers there exists the so called "scissors style", which is very popular in C++. In that style all fields are defined in the bottom section of a UDT.
  - In fact it doesn't matter: you should keep the fields always in a well known place.

Ultimate Restriction with C++ Classes

- Restricting the fields' accessibility of a UDT is called encapsulation.
  - The set of a UDT's public members is also called its interface.

- The fields contained in structs are publicly accessible by default.
  - After our experiences and efforts to have private fields we long for something better.
  - => We need a better means to get encapsulation.

```
struct Date { // (members hidden)
private:
	int month;
public:
	Date(int day, int month, int year);
	int GetMonth();
	void SetMonth(int month);
};
```

```
class Date { // (members hidden)
	int month;
public:
	Date(int day, int month, int year);
	int GetMonth();
	void SetMonth(int month);
};
```

- Instead of a struct we should use a class!
  - For now we can spot one difference between structs and classes:
    - All fields (and member functions) of a class are private by default.

- The compiler-generated dctor, cctor, copy-operator= and dtor are public and inline member functions also in classes!

The Class Date

```
// <main.cpp>
#include <iostream>

class Date {
	int day, month, year;
public:
	Date(int day, int month, int year);
	int GetDay();
	void SetDay(int day);
	int GetYear();
	void SetYear(int year);
	int GetMonth();
	void SetMonth(int month);
	void Print();
};
→
```

```
// → <main.cpp>
Date::Date(int day, int month, int year) {
	this->day = day;
	this->month = month;
	this->year = year;
}
int Date::GetDay() {
	return day;
}
void Date::SetDay(int day) {
	this->day = day;
}
int Date::GetMonth() {
	return month;
}
void Date::SetMonth(int month) {
	if (1<= month && month <= 12) {
		this->month = month;
	}
}
int Date::GetYear() {
	return year;
}
void Date::SetYear(int year) {
	return this->year = year;
}
void Date::Print() {
	std::cout<<day<<"."<<month<<"."<<year<<std::endl;
}
→
```

```
// → <main.cpp>

int main() {
	Date myDate(17, 10, 2012);
	myDate.Print();
	// >17.10.2012
}
```

Classes vs. Structs

- In short: classes and structs have the same features but different focuses.
  - C++' structs have **nothing** to do with C#'s structs or .NET's value types!
- When to use structs:
  - If we are programming in C instead of C++.
  - To define PODs, e.g. to read records from a file, i.e. for record-oriented programming.
  - To define PODs, creating objects to interact with the OS and hardware near devices.
  - Stringent definition of PODs: only fields of fundamental type, making PODs self-contained.
- When to use classes:
  - To define UDTs that abstract and encapsulate concepts.
- Rule: If there are no pressing needs prefer classes over structs for your UDTs.
  - (For completeness: In C/C++ we can also define unions as UDTs.)

- Self-contained PODs have no dependencies to other UDTs, which makes using them very straight forward.
- In future examples we'll stick to using classes instead of structs.

The Scope of Privacy

- private members can also be called on another instances of the defining UDT.
  - Assume the member function *Date::PrintMonth()* that accesses the passed *Date*.

```
void Date::PrintMonth(Date date) {
    // We can access the private field this->month, but we can also
    // access the private field month from other instances of Date:
    std::cout<<date.month<<std::endl; // Ok!
}
```

- But we shouldn't touch private fields on a regular basis: it breaks encapsulation!
  - Assume the member function *Date::ReadMonth()* that modifies the passed *Date**.

```
void Date::ReadMonth(Date* date) { // Dubious
    // If we set the field month directly, we
    // circumvented the checks we've introduced
    // with the member function SetMonth().
    std::cin>>date->month; // Hm... dubious!
    // What if the user entered 14?
}
```

```
void Date::ReadMonth(Date* date) { // Good
    // Better: call the encapsulating member
    // function w/ parameter checking.
    int month;
    std::cin>>month;
    date->SetMonth(month); // Good, will check
}                          // the passed value!
```

- We should always use setters or getters to modify or access fields.
  - Also in the defining UDT we should never access private fields directly!
  - Mind that esp. using the setters is important to exploit present parameter checking.

Inline Functions

- For some examples in this course inline definitions of member functions will be used.
  - We'll use inline code only because of brevity!

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

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

- The syntax of inline definitions of member functions:
  - Just define the complete member functions in the respective class definitions.

- Don't define inline member functions on a regular basis!
  - Sometimes inline member functions must be used in C++.
  - It can introduce problems in large projects, we'll discuss this topic in a future lecture.

Naming Conventions for abstracted Types

- Following naming conventions should be used in future.

- Names for UDTs:
  - Pascalcase
  - Don't use prefixes! (Like C for classes and S for structs.) Prefixes are pointless!

- Names for free functions and member functions:
  - Pascalcase

- Names for fields:
  - Camelcase

- Names for local variables:
  - Camelcase

Thank you!