

KIV/CPP – Programming in C++

05. Multiple inheritance, working with types, streams

Martin Úbl

KIV ZČU

2025/2026

Inheritance

- recap
- polymorphism (dynamic)
- keywords `virtual`, `override`
- inheritance visibility modifier
 - `public` – classic inheritance
 - `protected` – public and protected attributes and methods will be protected in the descendant
 - `private` – the same, but they will be private in the descendant

```
class Child : public Parent {  
    ...  
};
```

Virtual Method Table

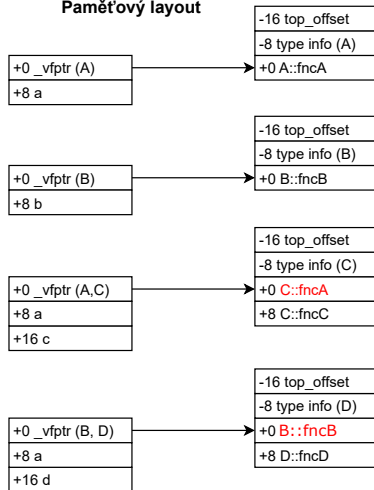
- a short digression: virtual method table
- a mechanism for implementing polymorphism
- essentially a list of addresses of virtual method implementations
- each descendant fills it in its own way
- the `virtual` keyword adds a method to the table
- contains additional service elements “above” the table itself
 - `top_offset` – offset of the table part from the top (when composing several tables into one large table)
 - `type_info` pointer – pointer to RTTI (see below)

- virtual method table and overridden methods

Definice třídy

```
class A {  
    public:  
        virtual void fncA();  
        int a;  
}  
  
class B {  
    public:  
        virtual void fncB();  
        int b;  
}  
  
class C : public A {  
    public:  
        virtual void fncA() override;  
        virtual void fncC();  
        int c;  
}  
  
class D : public B {  
    public:  
        virtual void fncD();  
        int d;  
}
```

Paměťový layout



Multiple Inheritance

- multiple inheritance
- it is possible, but sometimes unpredictable
- this can replace the absence of an *interface* language feature in combination with an abstract class
- a class can inherit from multiple parents, separated by commas in the definition
- it then really inherits from all parent classes

```
class Child : public Mother, public Father {  
    ...  
};
```

- multiple inheritance
- the memory layout is then “composed” together
 - virtual method tables as well
- it is necessary to account for the fact that a descendant can be cast to any of its ancestors
 - i.e., it must be possible to obtain a pointer to any of them without harm

Multiple Inheritance

- virtual method table and multiple inheritance

Definice třídy

```
class A {  
    public:  
        virtual void fncA();  
        int a;  
}
```

```
class B {  
    public:  
        virtual void fncB();  
        int b;  
}
```

```
class AB : public A, public B {  
    public:  
        virtual void fncA() override;  
        int x;  
}
```

Paměťový layout

+0 _vfptr
+8 a

-16 top_offset
-8 type info (A)
+0 A::fncA

+0 _vfptr
+8 b

-16 top_offset
-8 type info (A)
+0 B::fncB

+0 _vfptr (A)
+8 a
+16 _vfptr (B)
+24 b
+32 x

-16 top_offset
-8 type info (AB)
+0 AB::fncA
+8 top_offset
+16 type info (AB)
+24 B::fncB

- ... it brings problems with it
 1. what if multiple parent classes define a method with the same name?
 2. what if parent classes inherit from the same ancestor?
 - *diamond problem*
- can be avoided
 - by a better design (1, 2)
 - explicit ancestor specification when calling (1)
 - composition instead of polymorphism (1)
 - virtual inheritance (2)

Multiple Inheritance

- methods with the same name
- e.g., both Student and Employee have a GetTimeSchedule() method
 - however, each may do something slightly different

```
class WorkingStudent : public Student,  
                      public Employee {  
    ...  
};
```

- how to choose the correct version? explicitly

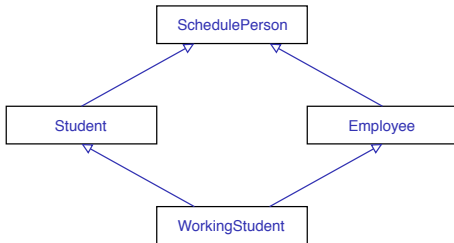
```
WorkingStudent pepa;
```

```
pepa.Student::GetTimeSchedule();  
pepa.Employee::GetTimeSchedule();
```

- methods overlap with each other
- try to avoid such a scheme
 - by a better design
 - by a different decomposition
 - by a common ancestor and virtual inheritance

Multiple Inheritance

- *diamond problem*
- the grandparent defines the `GetTimeSchedule` method including its implementation
- both parents inherit from the grandparent
- the descendant inherits from both parents and does not override `GetTimeSchedule`
- problem: both parents contain the grandparent's `GetTimeSchedule` in the virtual method table
 - which one is the correct one?



Multiple Inheritance

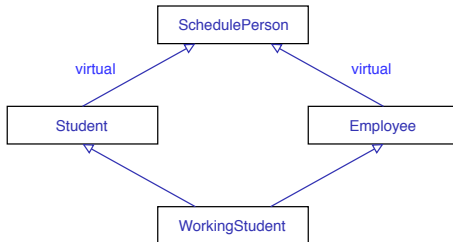
- problem: both parents contain the grandparent's `GetTimeSchedule` in the virtual method table
 - which one is the correct one?
- both, they are identical
- the problem is solved by virtual inheritance

```
class Student : public virtual SchedulePerson
{
    ...
};

class Employee : public virtual SchedulePerson
{
    ...
};
```

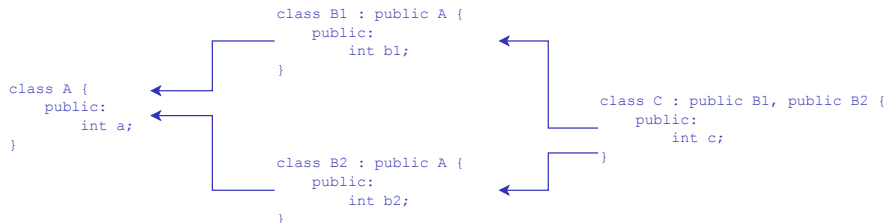
Multiple Inheritance

- *diamond problem*
- virtual inheritance of ancestors



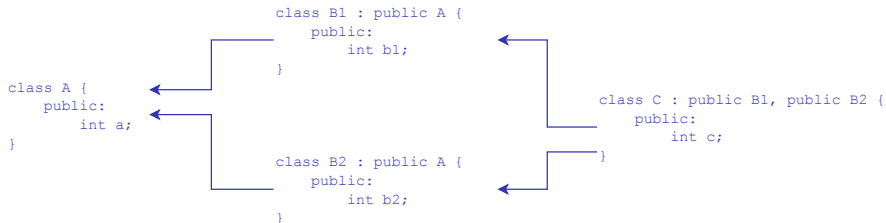
Multiple Inheritance

- *diamond problem*
- question: what will the memory layout look like?



Multiple Inheritance

- *diamond problem*
- question: what will the memory layout look like?



- without virtual inheritance, the layout of class A is duplicated

```
C test;  
test.a = 10; // ambiguous!
```

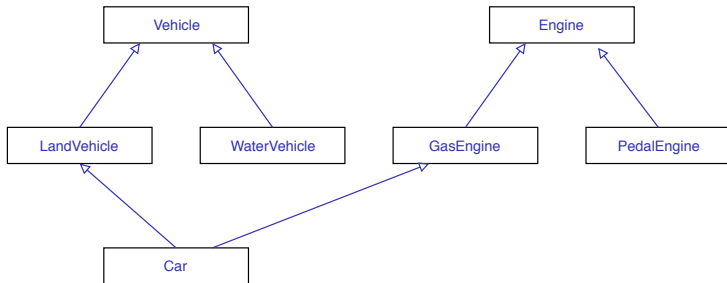
- myth:
 - multiple inheritance is bad!
- truth:
 - multiple inheritance is an irreplaceable tool for solving certain problems
 - always a question of design
 - it can be represented by static polymorphism (later seminars)

Multiple Inheritance

- useful when a descendant should share the ideas and behavior of several classes
 - sharing implementation
 - polymorphism
- “Rules of thumb”
 - <https://isocpp.org/wiki/faq/multiple-inheritance>
 - *Use inheritance only if doing so will remove if / switch statements from the caller code.*
 - *Try especially hard to use abstract base classes when you use multiple inheritance.*
 - *Consider the “bridge” pattern or nested generalization as possible alternatives to multiple inheritance.*
- note: *bridge pattern* – essentially composition with run-time type selection
- note 2: *nested generalization* – one primary hierarchy specialized for each subtype

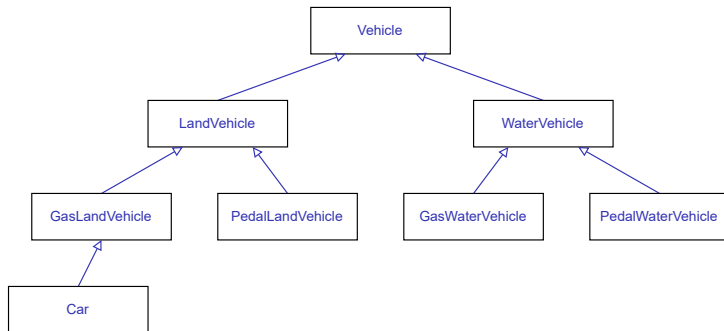
Multiple Inheritance

- isocpp example
- multiple inheritance and nested generalization



Multiple Inheritance

- isocpp example
- multiple inheritance and nested generalization



- working with types
- conversions between types
 - implicit
 - C-style cast
 - `static_cast`
 - `dynamic_cast`
 - `reinterpret_cast`
 - `const_cast`
- for polymorphic types in `shared_ptr`
 - `std::dynamic_pointer_cast`

- Run-Time Type Info (RTTI)
- information about an object's type at run time
- only for polymorphic types
- allows the use of some constructs
 - `dynamic_cast`
 - `typeid`
 - `type_info`
- the `type_info` item in the virtual method table refers to the RTTI instance

- `static_cast`
- “replacement” for implicit conversion
- “bypassing” narrowing conversion
- also, for example, when using C libraries – conversion between `void*` and a concrete pointer
- does not remove `const`

```
int a = 5;  
float b = static_cast<float>(a);  
int c = static_cast<int>(b);
```

```
void* mem = ...;  
uint8_t* cmem = static_cast<uint8_t*>(mem);
```

- `reinterpret_cast`
- forces the compiler to treat an entity as a completely different type
- can be dangerous – nothing stops us from converting, for example, an `int` to `std::string*`
- converting a pointer to a *float* with value `2.0f` to an *integer* does not give the value `2`
- relatively few situations where it is necessary
- *UserData* in libraries, converting structure to byte stream for networking, ...

```
float a = 2.0f;  
int b = *reinterpret_cast<int*>(&a);  
  
// b = 1073741824
```

- `dynamic_cast`
- `static_cast` with a check
- uses RTTI for conversion between polymorphic types
- casting a pointer to a pointer
 - returns `nullptr` on failure
- casting a reference to a reference
 - throws a `std::bad_cast` exception on failure

```
Parent *r = ...;  
Child *p = dynamic_cast<Child*>(r);  
  
if (p == nullptr)  
    // "r" is not of type Child
```

- `dynamic_cast`
- `reference`

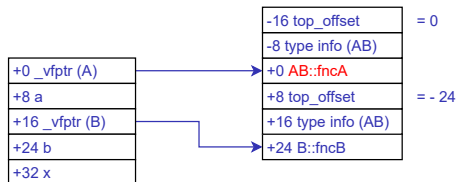
```
Parent &r = ...;
```

```
try
{
    Child& p = dynamic_cast<Child&>(r);
    p.method();
}
catch (const std::bad_cast& ex)
{
    // "r" is not of type Child
}
```

Working with Types

- `dynamic_cast`
- can return a different pointer
- see the memory layout for multiple inheritance
- `dynamic_cast` to different ancestors can return a different pointer
 - a reverse `dynamic_cast` to the descendant uses RTTI and the `top_offset` item in the virtual method table

```
class AB : public A, public B {  
    public:  
        virtual void fncA() override;  
        int x;  
}
```



- `const_cast`
- for adding or removing `const` from a type
- use very carefully when removing it
 - modifying constant memory can be undefined behavior
 - better to change the design

```
const char* pozdrav = "hello";
```

```
char* m_pozdrav = const_cast<char*>(pozdrav);  
m_pozdrav[0] = 'b'; // ouch!
```

- question: why would this modification likely fail?

- `const_cast`
- memory that is not actually const, but is passed as const
- e.g., in `std::string`
 - `c_str()` returns `const char*`

```
std::string s_pozdrav = "hello";
```

```
char* ms_pozdrav = const_cast<char*>  
                    (s_pozdrav.c_str());  
ms_pozdrav[0] = 'b';
```

Summary

- C-style cast
 - forget about it
- `static_cast`
 - for ordinary casts
 - POD types among themselves, objects if we do not need RTTI
- `dynamic_cast`
 - for polymorphic casts
- `reinterpret_cast`
 - avoid if possible
 - otherwise, for example, when raw memory work is necessary
- `const_cast`
 - preferably never
 - ideally only for adding `const`
 - remove `const` only if there really is no other option

- the `decltype` keyword
 - determining and applying a type (e.g., to ensure that the type will be identical)

```
unsigned long long a = 15;  
decltype(a) b = 25;
```

- the `typeid` keyword
 - returns RTTI information in a `std::type_info` instance
 - can be useful, for example, for debugging

```
std::string p;  
std::cout << typeid(p).name();
```

- a general mechanism for working with input-output streams
- input stream
 - `std::istream`
- output stream
 - `std::ostream`
- combination
 - `std::iostream`
- specializations for files and strings
- overloaded bit-shift operators `<<` and `>>`
 - only for text input and output!

- respects RAI
 - the constructor opens the stream (e.g., a file)
 - the destructor closes the stream
- open flags (always `std::ios::`)
 - `in` – we will read from the stream ("r")
 - `out` – we will write to the stream ("w")
 - `app` – the cursor will move to the end of the stream ("a")
 - `trunc` – the stream is emptied when opened
 - `binary` – binary reading/writing ("b")

- file streams
- `#include <fstream>`
- input stream
 - `std::ifstream`
- output stream
 - `std::ofstream`
- combination
 - `std::fstream`
- constructor parameters
 - path to the file
 - optionally the open mode

- file streams
- output text stream

```
std::ofstream output("file.txt");
```

```
output << "Hello" << std::endl;
```

```
output << "World" << std::endl;
```

- file streams
- input text stream
- the implicit delimiter is a space or line ending

```
std::ifstream input("file.txt");
```

```
std::string a, b;
```

```
input >> a;
```

```
input >> b;
```

- file streams
- binary writing

```
std::ofstream output("file.bin",  
                    std::ios::out | std::ios::binary);  
  
const char* binbuf = "Binary\x00output\x00";  
  
output.write(binbuf, 14);
```

- file streams
- binary reading

```
std::ifstream input("file.bin",  
                    std::ios::in | std::ios::binary);
```

```
char data[14];
```

```
input.read(data, 14);
```

String Streams

- string streams
- `#include <sstream>`
- input stream
 - `std::istringstream`
- output stream
 - `std::ostringstream`
- combination
 - `std::stringstream`
- constructor parameters
 - string to use as the base
 - optionally also the open mode
- the content is then retrieved with the `str()` method
- otherwise it behaves the same as a file stream

String Streams

- string streams
- output string stream

```
std::ostream output(  
    "On_the_seventh_day_God_said:",  
    std::ios::app);
```

```
output << "Hello";  
output << "_";  
output << "World";
```

```
std::string result = output.str();
```

- note: without the *append* flag, the base text would be overwritten

- `std::getline`
- obtains the whole “line”
 - otherwise parsing also stops at spaces by default
- it is possible to define the character to “stop at”
- it also returns a stream instance, which has an overloaded `bool()` operator
 - the success of the operation can therefore be checked easily

```
std::ifstream input("file.txt");
```

```
std::string str;  
while (std::getline(input, str))  
    std::cout << str << std::endl;
```

- `std::getline`
- e.g., parses by semicolons

```
std::ifstream input("file.csv");
```

```
std::string str;  
while (std::getline(input, str, ';'))  
    std::cout << str << std::endl;
```

- *note: of course, this is not a complete CSV parser*

- determining the position in a stream
 - `tellg` (input), `tellp` (output)
 - returns a `streampos` instance (often `int` or `size_t`)
- setting the position in a stream (`seek`)
 - `seekg` (input), `seekp` (output)
- e.g., determining the size of a file in bytes:

```
std::ifstream input("file.txt");
```

```
input.seekg(0, input.end);  
int fileSize = input.tellg();  
input.seekg(0, input.beg);
```

- standard streams
 - stream wrappers over implicitly opened files
 - `std::cin` – standard input, `stdin`
 - `std::cout` – standard output, `stdout`
 - `std::cerr` – error output, `stderr`
- stream instances can be passed by a general reference
- it is therefore possible to have a function capable of writing to/reading from any stream
 - parameter `std::ostream&`
 - passing `std::cout` writes to the console
 - passing a `std::ofstream` instance writes to a file
 - parameter `std::istream&`
 - passing `std::cin` reads from the console
 - passing a `std::ifstream` instance reads from a file

- special input/output “characters”
- `std::endl`
 - inserts a line ending `\n` and calls `flush()`
 - note: `flush()` invokes a *syscall* (can slow things down)
- `std::ends`
 - inserts a null character `\0`
- `std::ws`
 - “discards” whitespace (space, tab) from input

Stream Manipulators

- stream manipulators
- `#include <iomanip>`
- contain modifiers for how stream input/output is processed
- e.g., number format
 - hex, dec, oct
 - fixed, scientific (floating-point)
- number width and fill
 - `setprecision`
 - `setw`, `setfill`
- type transcription
 - `boolalpha` and `noboolalpha`
- and others
 - `showbase` and `noshowbase`
- modifiers are “inserted” into a stream by the `<<` and `>>` operators

- stream manipulators
- e.g., hex output

```
std::cout << std::showbase << std::hex << 254;  
// 0xfe
```

- there are many of them
- <https://en.cppreference.com/w/cpp/io/manip>
- more in examples

- `std::format` (since C++20)
- `#include <format>`
- counterpart of similar formatting functions from other languages
- allows output to be formatted based on a format string
 - placeholder sequences in `{ ... }`
 - optionally a parameter index, colon, and formatter inside
 - escaping by doubling
- uses variadic templates for a variable number of parameters (see later lectures)

```
std::format("Hello_{}!", "world");  
std::format("Hello_{1}_and_{2}!_",  
            "Atlas", "P-body");  
std::format("Number_{:.2f}", 15.245);
```

- `std::format`, examples of formatting options
- `{:#x}`, `{:#o}`, `{:#b}` – hexadecimal, octal, and binary output, including the prefix (#)
- `{:0>6}`, `{:0<6}`, `{:0^6}` – right alignment, left alignment, centering, padding to 6 characters, zero fill
 - 000066, 660000, 006600
- and more...

- `std::format` (since C++20)
- there are many formatters; they resemble those from C (`printf`)
- `https://en.cppreference.com/w/cpp/utility/format/format`
- more in the seminar

- `std::vformat` (since C++20)
- dynamic formatting
 - throws `std::format_error` on invalid formatter string
 - unlike `std::format`, which is completely statically checked
- e.g., formatting a list of names with alignment to the longest one

```
std::string fmt = std::format("{:{>{}}}",  
                               longest.length());  
  
for (const auto& p : osoby) {  
    std::cout  
        << std::vformat(fmt, std::make_format_args(p))  
        << std::endl;  
}
```

- `std::format` (since C++20)
- supports recursive definitions
- a format defined by a format
- see the previous example, but better, using only `std::format`

```
for (const auto& p : osoby) {  
    std::cout  
        << std::format("{:>{}}", p, longest.length())  
        << std::endl;  
}
```

- `std::print` and `std::println` (since C++23)
- formatting text directly to standard output
- format identical to `std::format`

```
std::println("So we finally got to {}!", "the end");
```

```
std::println("First five even numbers: {n}",  
    std::views::iota(0, 5) |  
    std::views::transform([](int v) {  
        return (v + 1) * 2;  
    })  
);
```

- C++ streams are not necessarily the most efficient option
 - an “intermediate point” of readability, universality, and performance
 - it is possible to optimize for one thing or another
- streams are not thread-safe
 - e.g., writing to `std::cout` from multiple threads in multiple parts tears and mixes the output in various ways
 - partially solved by `std::ostream`
- in modern C++, it is almost always better to use `std::print` or `std::format` in combination

- C++ streams for binary reading and writing
- KIV/ZOS
 - implementation of a virtual FS
 - binary file as a disk image
 - need to read and write structures
 - padding, alignment
 - endianness
 - binary data
- KIV/UPS
 - possible use of streams for working with application packets
 - text protocol, stringstream
 - since C++2a, possibly also a network stream
- examples in the seminar